Theoretical Computer Science 28 (1984) 45–81
North-Holland

PARAMETER PASSING IN ALGEBRAIC SPECIFICATION LANGUAGES*

Hartmut EHRIG and Hans-Jörg KREOWSKI

Technische Universität Berlin, Fachbereich Informatik 20, Institut für Software und Theoretische Informatik, D-1000 Berlin 10, Fed. Rep. Germany

James THATCHER, Eric WAGNER and Jesse WRIGHT

IBM Research Center, Mathematical Sciences Department, Yorktown Heights, NY 10598, U.S.A.

Communicated by R. Milner
Received September 1981
Revised January 1983

Abstract. In this paper we study the semantics of the parameter passing mechanism in algebraic specification languages. More precisely, this problem is studied for parameterized data types and parameterized specifications. The given results include the extension of the model functor (which is useful for correctness proofs) and the semantic properties of the result of inserting actual parameters into parameterized specifications. In particular, actual parameters can be parameterized and the result is nested parameterized specification. Correctness of an applied (**matrix**(**int**)) or a nested (**bintree**(**string**( ))) parameterized specification is shown given correctness of the parts. The formal theory in this paper is restricted to the basic algebraic case where only equations are allowed in the parameter declaration and parameter passing is given by specification morphisms. But we also give the main ideas of a corresponding theory with requirements where we allow different kinds of restrictions in the parameter declaration.

1. Introduction

Procedural abstraction has been with us a long time both in practice and in theory, although the semantic theory for procedures taking procedures as parameters is relatively recent (c.f. Scott [38]). A practical analog of procedural abstraction for data definition is relatively new (for example, see [23, 25, 34, 35, 40]). The semantic theory for parameterized types is the subject of this paper. There has been little work on the mathematics of parameter passing with the exception that Burstall and Goguen have tackled it for the mathematical semantics of CLEAR because procedures

* A short and slightly different version of this paper has appeared as "Parameterized data types in algebraic specification languages" [5] in the Proceedings of 7th ICALP, Noordwijkerhout, 1980. The present paper is a restriction of our March 1980 draft version to the basic algebraic case. However, it is also the basis for a theory of parameterized specifications with requirements (see [13]) which avoids the concept of generalized parameter passing in [5].

A preliminary version of this paper has appeared in the Proceedings of the Aarhus Workshop on Program Specification.

in CLEAR correspond to parameterized types [7, 8, 9]. Also Ehrich [11] and Ehrich and Lohberger [12] study parameterization on a syntactic level, as a relationship between specifications. Although the ADJ-Group [4] provides us with an algebraic formulation for parameterized types, they barely touch the question of parameter passing.

The problem of parameter passing for data abstractions is, we believe, an important one. Hierarchical design of large programming systems depends on the use of parameterized data abstractions (even familiar **string( )**, **array( )** or **structure( )**) and an understanding of the semantics of parameter passing is a prerequisite to the understanding of the mathematical semantics of the hierarchical design.

In this paper we do several things. First of all, we give a precise mathematical definition of what it means to insert a parameter into a parameterized type (e.g., inserting **int** into **string( )**). Our approach is sufficiently general that it provides the necessary apparatus for approaching many related problems, e.g., the inserting of non-parameterized specifications into parameterized specifications, the composition of parameterized types or specifications, the compatibility of different 'call by name' strategies, compatibility of 'call by name', and 'call by value', proofs of correctness [e.g., that if we have a correct specification for **int** and **string( )** then this implies the correctness of the specification **string(int)**], etc. We will treat all of these in detail within the present paper while a short and slightly different version including only part of the results is given in our proceedings version [5].

Following the general frame of our paper on algebraic implementations [18] syntax, semantics, semantical requirements and correctness of parameter passing are carefully distinguished, motivated and studied in detail within Section 2.

In Section 3 the parameterized types **string(param)**, **matrix(ring)** and **bintree(data)** are specified as typical examples for the basic algebraic case studied in this paper. For examples like **set(data)** using non-equational requirements in the parameter declaration we refer to [13].

In Section 4 we start with the simplest case of 'standard parameter passing' where actual specifications like **int** are inserted for formal parameters like **param** in **string(param)** leading to actual value specifications like **string(int)**. Corresponding correctness results are given in Section 5. The key for the proofs is Extension Lemma 5.1 which was motivated by corresponding constructions in [4, 11, 12]. In addition to the construction of pushouts in the category of specifications, which is also considered in [11, 12, 8, 9] we also have an explicit and a universal construction for the semantics of the value specification, which for a special case was considered in [4]. Extension Lemma 5.1 is sufficient to show correctness of standard parameter passing in Theorem 5.2 and induced correctness of value specifications in Theorem 5.4.

In Section 6 we study 'parameterized parameter passing' where the actual parameter to be inserted is again a parameterized specification. Hence the value specification becomes a parameterized specification like **bintree(string(param))**. This parameterized value specification can also be considered as the composition of the

parameterized specifications **bintree(data)** and **string(param)**. The theory of parameterized parameter passing is not much different from that of standard parameter passing. A slight modification of results and proofs shows that also parameterized parameter passing is correct (Theorem 6.3) and that we have induced correctness of composite parameterized specifications (Theorem 6.4).

Iterated compositions and different evaluation strategies built up by standard parameter passing and parameterized parameter passing steps are studied in Section 7. Associativity of the composition (Theorem 7.1) and compatibility of composition and actualization (Theorem 7.2) shows that the result of iterated parameter passing is independent of the choice of the evaluation strategy (Corollary 7.4).

In contrast to our short version [5] this paper is restricted to the basic algebraic case where instead of universal Horn sentences we only use equations in all our specifications including the parameter declaration. Moreover, parameter passing is defined using specification morphisms which automatically imply passing consistency in the sense of [5]. Due to these restrictions a number of interesting parameterized specifications like **set(data)**, **stack(attr)** and **queue(items)** are excluded if non-equational requirements are used to define an equality predicate in the parameter declaration. On the other hand, the mathematical theory in the basic algebraic case is much simpler than that in [5] and can be fully extended to an algebraic theory of parameterized specifications with requirements (see [13]). Requirements in our sense were motivated and include those of Burstall and Goguen and Hupbach and Reichel which are called Σ-constraints in [9] and initial restrictions in [37]. Some basic results and examples of our theory with requirements are given in [13] and [14] but the complete theory is still in development. A summary of the present paper, the main ideas of our theory with requirements and a comparative discussion of other approaches are given in our conclusion (Section 8).

2. Parameterized types and specifications

We shall assume the algebraic background of [3, 17, 33] which is based on universal algebra (see [10, 24]) and category theory (see [6, 28, 36]). But we will review the most important notions in connection with this paper. We shall introduce the basic algebraic case of parameterized data types and specifications as given in [4]. An *abstract data type* is regarded as (the isomorphism class of) a many-sorted (heterogeneous) algebra which is minimal, meaning that all data elements are 'accessible' using constants and operations of the algebra. A many-sorted algebra consists of an indexed family of sets (called *carriers*) with an indexed family of operations between those carriers. The indexing system is called a *signature* and consists of a set S of *sorts* which indexes the carriers and a family $(\Sigma_{w,s} \mid w \in S^*$ and $s \in S)$ of operation names (Σ is called the *operator domain*); a symbol $\sigma \in \Sigma_{w,s}$ with $w = s1 \cdots sn$ names an operation $\sigma_A: A_{s1} \times \cdots \times A_{sn} \to A_s$ in an algebra A with

signature Σ. The pair $\langle S, \Sigma \rangle$ determines the category $Alg_{\langle S,\Sigma \rangle}$ of all S-sorted Σ-algebras with Σ-homomorphisms between them.

A *specification*, SPEC $= \langle S, \Sigma, E \rangle$, is a triple where $\langle S, \Sigma \rangle$ is a signature and E is a set of equations. Alg_{SPEC} is the category of all SPEC-algebras, i.e., all S-sorted Σ-algebras satisfying the equations E. When we write the *combination* SPEC' = SPEC $+ \langle S', \Sigma', E' \rangle$ we mean that S and S' are disjoint, that Σ' is an operator domain over $S + S'$ which is disjoint from Σ, and that E' is a set of axioms over the signature $\langle S + S', \Sigma + \Sigma' \rangle$.

Although some authors see the equations as 'semantics' (see [25, 26]), we follow [3] in saying that the *semantics* of a specification SPEC is the (isomorphism class of the) algebra T_{SPEC} which is initial in Alg_{SPEC}. T_{SPEC} can be constructed as a quotient $T_{\text{SPEC}} = T_{\langle S,\Sigma \rangle} / \equiv_E$ of the term algebra $T_{\langle S,\Sigma \rangle}$ (corresponding to the signature $\langle S, \Sigma \rangle$) by the congruence generated from the equations E.

A specification SPEC $= \langle S, \Sigma, E \rangle$ is called *correct* with respect to a model algebra A in Alg_{MSPEC} if the model specification MSPEC $= \langle MS, M\Sigma, ME \rangle$ is included in SPEC, i.e., $MS \leq S$, $M\Sigma \leq \Sigma$ and $ME \leq E$, and the MSPEC-reduct of T_{SPEC} is isomorphic to A. (The MSPEC-reduct of T_{SPEC} consists of those carriers and operations belonging to sorts of MS and operation symbols of $M\Sigma$ respectively.) Note that this definition allows to use 'hidden functions' which are included in the specification but not in the model specification. Moreover, in most cases ME will be the empty set of equations.

Now let us consider parameterized data types and specifications.

2.1. Definition. A *parameterized data type* PDAT $=$ (SPEC, SPEC1, T) consists of the following data:

PARAMETER DECLARATION SPEC $= (S, \Sigma, E)$

TARGET SPECIFICATION SPEC1 = SPEC $+ \langle S1, \Sigma 1, E1 \rangle$

and a functor $T: Alg_{\text{SPEC}} \to Alg_{\text{SPEC1}}$. PDAT is called *persistent* (*strongly persistent*) if T is, i.e., for every SPEC-algebra A we have a natural isomorphism $V(T(A)) \cong A$ (resp. $V(T(A)) = A$) where V is the forgetful functor from SPEC1 to SPEC-algebras (see Proposition 4.2).

Remark. Although not explicitly used here, we assume that T is equipped with a (natural) family of homomorphisms $\langle I_A : A \to V(T(A)) \rangle$ such that for each SPEC-algebra A, the set $\{I_A(a) \mid a \in A\}$ generates $T(A)$. (This generalizes the minimality condition in [4].) As discussed in [4], the family I tells how to find each parameter algebra A in the result of the construction $T(A)$. The motivation for persistence is given in [4]; the idea is that the parameter algebra 'persists' (up to isomorphism) in the result of the construction T. In contrast to [4] and [5] we only allow equations in specifications. Negative conditional and universal Horn axioms may be included in requirements (see Section 8).

To illustrate our definitions we will construct the model functor SET0 corresponding to a simplified version **set0(data0)** of a set specification where only the operations CREATE and INSERT are considered. In this simplified version we do not need the equality predicate and hence also not **bool** in the parameter declaration.

2.2. Example

PARAMETER DECLARATION (SPEC = $\langle S, \emptyset, \emptyset \rangle$) : **data0** =
  sorts(S) : *data*
TARGET SPECIFICATION (MSPEC1 = SPEC + $\langle S1, \Sigma 1, \emptyset \rangle$) : **Mset0** =
  data0 +
  sorts($S1$) : *set*
  opns($\Sigma 1$) : CREATE : → *set*
  INSERT : *data set* → *set*

The model functor SET0: $Alg_{\mathbf{data0}} \to Alg_{\text{Mset0}}$ takes each **data0**-algebra $E = \langle E_{data} \rangle$, which is simply a set of parameter elements, to the **Mset0**-algebra $A = \langle A_{data}, A_{set}, \text{CREATE}_A, \text{INSERT}_A \rangle$ with $A_{data} = E_{data}$, $A_{set} = \mathcal{P}_{\text{fin}}(E_{data})$ the set of all finite subsets of E_{data}, $\text{CREATE}_A = \emptyset$ and $\text{INSERT}_A(e, s) = \{e\} \cup s$ for all $e \in E_{data}$ and $s \in \mathcal{P}_{\text{fin}}(E_{data})$. The model functor is strongly persistent because we have $V(\text{SET0}(E)) = V(A) = A_{data} = E$. In the following we shall show that the simplified version **set0(data0)** of our set specification is correct with respect to the parameterized model data type PMDAT = ⟨**data0**, **Mset0**, SET0⟩ defined above.

2.3. Definition. (1) A *parameterized specification* PSPEC = ⟨SPEC, SPEC1⟩ consists of the following data:

PARAMETER DECLARATION SPEC = $\langle S, \Sigma, E \rangle$
TARGET SPECIFICATION SPEC1 = SPEC + $\langle S1, \Sigma 1, E1 \rangle$

The semantics of the specification is the free construction (see [4]), $F: Alg_{\text{SPEC}} \to Alg_{\text{SPEC1}}$, i.e., the parameterized type PDAT = ⟨SPEC, SPEC1, T⟩.

Remark. We will talk about the 'parameterized type ⟨SPEC, SPEC1⟩' and mean the type whose (model) functor is the free construction from SPEC-algebras to SPEC1-algebras.

(2) Let PDAT = ⟨MSPEC, MSPEC1, T⟩ be a parameterized data type and PSPEC = ⟨SPEC, SPEC1⟩ a parameterized specification. Then PSPEC is called *correct with respect to* PDAT if we have MSPEC ⊆ SPEC, MSPEC1 ⊆ SPEC1 and (up to isomorphism) $T \circ U = U1 \circ F$ with surjective forgetful functor $U: Alg_{\text{SPEC}} \to Alg_{\text{MSPEC}}$, forgetful functor $U1: Alg_{\text{SPEC1}} \to Alg_{\text{MSPEC1}}$ and $F: Alg_{\text{SPEC}} \to Alg_{\text{SPEC1}}$ the semantics (free construction) of PSPEC.

Remark. If U and $U1$ are identity functors correctness means that the free construction F is equal to the given model functor T. Otherwise they have to be equal up to renaming and forgetting of those sorts and operations which are in SPEC1 but not in MSPEC1. Surjectivity of U (which is not assumed in [4] and [5]

makes sure that for each model parameter algebra in Alg_{MSPEC} there is also a corresponding parameter algebra in Alg_{SPEC}.

2.4. Fact. *The parameterized specification* **set0(data0)** *– given by* **data0** *and* **set0** = **Mset0** + $E1$ *as in Example* 2.2 *with* $E1$ *consisting of the equations* INSERT(d, INSERT(d, s)) = INSERT(d, s) *and* INSERT(d, INSERT(d', s)) = INSERT(d', INSERT(d, s)) *– is correct with respect to the parameterized type* PMDAT = ⟨**data0**, **Mset0**, SET0⟩ *(see Example* 2.2).

Proof. In our case U and $U1$ are identity functors and it remains to show that the functor SET0 considered as functor from $Alg_{\textbf{data0}}$ to $Alg_{\textbf{set0}}$ is the free construction with respect to the forgetful functor $V: Alg_{\textbf{set0}} \to Alg_{\textbf{data0}}$. This means that we have to show for each $B \in Alg_{\textbf{set0}}$ and each homomorphism $f: E \to V(B)$ that there is a unique **set0**-morphism $g: \text{SET0}(E) \to B$ with $g_{data} = f_{data}$. (The homomorphism $f: E \to V(B)$ has one component $f_{data}: E_{data} \to V(B)_{data}$ while $g: \text{SET0}(E) \to B$ has two, $g_{data}: \text{SET0}(E)_{data} \to B_{data}$ and $g_{set}: \text{SET0}(E)_{set} \to B_{set}$. The crucial step here is the definition of the *set*-component of g.) Since g must be a **set0**-morphism we have for the *set*-component of g,

$$g(\emptyset) = g(\text{CREATE}_{\text{SET0}(E)}) = \text{CREATE}_B$$

and

$$g(\{e\} \cup s) = g(\text{INSERT}_{\text{SET0}(E)}(e, s)) = \text{INSERT}_B(f(e), g(s)).$$

Using the INSERT-equations for INSERT_B it is easy to show that the equations above define a well-defined **set0**-morphism $g: \text{SET0}(E) \to B$ with $g_{data} = f_{data}$. □

Now we want to consider a more complete version **set(data)** of a set specification which also includes the operations DELETE, MEMBER and EMPTY. In [4] it is shown that this also implies that we need an equality predicate EQ on **data**. To express that EQ is really an equality predicate needs a negative conditional axiom like

$$X \neq Y \Rightarrow \text{EQ}(X, Y) = \text{FALSE}$$

which is not allowed in the basic algebraic case. Hence we only consider the equation EQ(X, X) = TRUE at the moment while the general case with requirements is discussed in Section 8.

2.5. Example (**set(data)** without requirements)

PARAMETER DECLARATION: **data** =
  bool +
  sorts: *data*
  opns: *EQ*: *data data* → *bool*
  eqns: *EQ*(X, X) = TRUE
TARGET SPECIFICATION: **set(data)** =
  data +

sorts: *set*

opns: CREATE: → *set*
INSERT: *data set* → *set*
DELETE: *data set* → *set*
MEMBER: *data set* → *bool*
EMPTY: *set* → *bool*
IF-THEN-ELSE: *bool set set* → *set*

eqns: INSERT(d, INSERT(d', s)) = IF EQ(d, d') THEN
INSERT(d, s)ELSE INSERT(d', INSERT(d, s))
DELETE(d, CREATE) = CREATE
DELETE(d, INSERT(d', s)) = IF EQ(d, d') THEN
DELETE(d, s)ELSE INSERT(d', DELETE(d, s))
MEMBER(d, CREATE) = FALSE
MEMBER(d, INSERT(d', s)) = IF EQ(d, d') THEN TRUE
ELSE MEMBER(d, s)
EMPTY(CREATE) = TRUE
EMPTY(INSERT(d, s)) = FALSE
IF TRUE THEN $s1$ ELSE $s2 = s1$
IF FALSE THEN $s1$ ELSE $s2 = s2$

where **bool** is some correct specification of boolean values including TRUE, FALSE, AND, OR, NON and IF-THEN-ELSE operations.

In the semantics of this specification, however, we can also use parameter algebras A where A_{bool} has more than two elements. In this case the free construction $F(A)$ generates via the IF-THEN-ELSE-operations new data of sort *set* which cannot be generated by CREATE or INSERT. This implies that also new data of sort *bool* are generated via the operations MEMBER and EMPTY which means $F(A)_{bool} \neq A_{bool}$ such that F is not persistent and is not correct with respect to SET0 in Example 2.2. This is a common error in parameterized set specifications (see [4, 5]) which can be avoided using initial restrictions (see [13] and Section 8).

3. The parameterized types string, matrix and bintree

In this section we give some more examples of parameterized types which can be handled within the basic algebraic approach where in the formal parameter part only equations are allowed. In Example 2.5 we have seen that the parameterized type **set(data)** cannot be handled nicely in the basic algebraic approach because we need an equality predicate and initiality of the **bool**-part for all formal parameters. This can only be achieved in the case of parameterized specifications with requirements which will be sketched in Section 8. In this more general setting we can also handle parameterized types like **stack(attr)**, **queue(par)** and **array(item)** which also use an equality predicate with initial **bool**-part and, in addition, error handling.

For some specifications, however, like **string(param)**, it is possible to avoid the requirement that the **bool**-part is initial and that EQ: *data data* → *bool* is really an equality predicate. This specification makes also sense in the case that EQ is only a reflexive relation as well as when the **bool**- and **nat**-parts are not initial; these more general cases do not violate the persistency of the semantics of the specification.

In the following we give the parameterized specifications for **string(param)**, **matrix(ring)**, **bintree(data)** and **bintreetraversal(data)** and a few remarks concerning the semantics. For complete semantical models and corresponding correctness proofs in the sense of Definition 2.3(2) we refer to [19].

3.1. Example (string(param)

PARAMETER DECLARATION: **param** =
nat + **bool** +
sorts : *data*
opns : EQ: *data data* → *bool*
eqns : EQ(X, X) = TRUE
Comment: We only require reflexivity and not equality or an equivalence relation

TARGET SPECIFICATION: **string(param)** =
param +
sorts: *string*
opns: EMPTY: → *string*
LETTER: *data* → *string*
CONCAT: *string string* → *string*
LADD: *data string* → *string*
RADD: *string data* → *string*
ROTATE: *string* → *string*
REVERSE: *string* → *string*
SHUFFLE: *string string* → *string*
LENGTH: *string* → *nat*
IS-EMPTY: *string* → *bool*
IS-IN: *data string* → *bool*
eqns: CONCAT(S, EMPTY) = CONCAT(EMPTY, S) = S
CONCAT(CONCAT(S, S'), S'') = CONCAT(S, CONCAT(S', S''))
LADD(D, S) = CONCAT(LETTER(D), S)
RADD(S, D) = CONCAT(S, LETTER(D))
ROTATE(EMPTY) = EMPTY
ROTATE(LADD(D, S)) = RADD(S, D)
REVERSE(EMPTY) = EMPTY
REVERSE(LADD(D, S)) = RADD(REVERSE(S), D)
SHUFFLE(EMPTY, S) = SHUFFLE(S, EMPTY) = S

SHUFFLE(LADD(D, S), LADD(D', S'))
= LADD(D, LADD(D', SHUFFLE(S, S')))
LENGTH(EMPTY) = 0
LENGTH(LADD(D, S)) = SUCC(LENGTH(S))
IS-EMPTY(EMPTY) = TRUE
IS-EMPTY(LADD(D, S)) = FALSE
IS-IN(D, EMPTY) = FALSE
IS-IN(D, LADD(D', S)) = EQ(D, D') OR IS-IN(D, S)

The parameter declaration **param** consists of a specification **nat** for natural numbers with zero (0) and successor (SUCC), **bool** with boolean operations AND and OR, a sort *data* and a reflexive relation EQ on *data*. Hence **param**-algebras are 3-sorted where the **nat**- and **bool**-parts are (not necessary initial) **nat**- resp. **bool**-algebras and the *data*-part is a set with a reflexive relation. Even if the **bool**-part is not two-valued, the relation can be defined as preimage of TRUE. The semantics of **string(param)** is the free construction F which is persistent and assigns to each **param**-algebra A the free algebra $F(A)$ where $F(A)_{string}$ is the free monoid A^*_{data} containing all strings over the alphabet A_{data}. EMPTY creates the empty string, LETTER creates for each data a the corresponding string of length 1, CONCAT is concatenation of strings, LADD (resp. RADD) is left (resp. right) addition of a data to a string, ROTATE is rotation of the string by one position to the left where the first letter becomes the last, REVERSE reverses the string, SHUFFLE of two strings constructs the shuffle product, LENGTH measures the length of each string which, however, counts modulo m if A_{nat} is isomorphic to $\mathbb{N}$ (mod m), IS-EMPTY and IS-IN are predicates assigning the value TRUE if the string is empty resp. the string contains an element EQ-related (or equal if EQ is the equality on data) to the given one.

3.2. Example (matrix(ring)). For simplicity we only consider 2×2-matrices:

PARAMETER DECLARATION: **ring** =
sorts: *ring*
opns: $0, 1 : \to ring$
$+, * : ring\ ring \to ring$
$- : ring \to ring$
eqns: $(X + Y) + Z = X + (Y + Z)$
$X + Y = Y + X$
$X + (-X) = 0$
$X + 0 = X$
$(X * Y) * Z = X * (Y * Z)$
$X * Y = Y * X$
$X * 1 = X$
$X * (Y + Z) = X * Y + X * Z$

TARGET SPECIFICATION: **matrix(ring)** =
ring +

sorts: *matrix*

opns: ZERO, UNIT: → *matrix*

MATRIX: *ring ring ring ring* → *matrix*

ADD,SUB,MUL: *matrix matrix* → *matrix*

DET: *matrix* → *ring*

eqns: ZERO = MATRIX(0, 0, 0, 0)

UNIT = MATRIX(1, 0, 0, 1)

ADD(MATRIX($A1, A2, A3, A4$), MATRIX($B1, B2, B3, B4$))
= MATRIX($A1+B1, A2+B2, A3+B3, A4+B4$)

SUB(MATRIX($A1, A2, A3, A4$), MATRIX($B1, B2, B3, B4$))
= MATRIX($A1+(-B1), A2+(-B2), A3+(-B3),$
$A4+(-B4)$)

MUL(MATRIX($A1, A2, A3, A4$), MATRIX($B1, B2, B3, B4$))
= MATRIX($A1*B1+A2*B3, A1*B2+A2*B4,$
$A3*B1+A4*B3, A3*B2+A4*B4$)

DET(MATRIX($A1, A2, A3, A4$)) = $A1*A4+(-A2*A3)$

The parameter declaration **ring** corresponds to the usual axiomatic definition of commutative rings with unit. Hence a **ring**-algebra is an arbitrary commutative ring with unit while the initial **ring**-algebra is isomorphic to the ring of integers. The semantics of **matrix(ring)** is the free construction F which is persistent and assigns to each ring R the **matrix(ring)**-algebra $F(R)$ where $F(R)_{matrix}$ is the set of all 2×2-matrices over R with zero matrix ZERO, unit matrix UNIT, addition ADD, subtraction SUB and multiplication MUL of matrices while DET calculates the determinant of a matrix. This example could be extended by an operation ISSING: *matrix* → *bool* testing singularity of matrices provided that the parameter declaration is extended by **bool** and an equivalence relation EQ on **ring** which corresponds to the equality for a given ring R. If we want to make sure that EQ is the equality on **ring** we need similar requirements as for EQ on *data* in **set(data)** (see Section 8).

3.3. Example (**bintree(data)** and **bintreetraversal(data)**)

(1) We first give a parameterized specification for binary trees:

PARAMETER DECLARATION: **data** =

sorts: *data*

TARGET SPECIFICATION: **bintree(data)** =

data +

sorts: *bintree*

opns: LEAF: *data* → *bintree*

LEFT,RIGHT: *data bintree* → *bintree*

BOTH: *data bintree bintree* → *bintree*

This skeleton specification of binary trees which has no equations can be enriched by typical tree operations like HEIGHT, BREADTH, NODES, EDGES, BAL and DEG

measuring the height, the number of leaves, nodes and edges of the tree and testing whether it is balanced resp. degenrated (i.e., without branching). However, such an enrichment in the target specification needs an extension of the parameter declaration by a **nat**- and a **bool**-part. But unlike **set(data)** in Example 2.5 initiality of **nat** and **bool** is not necessary to show persistency and correctness (see [19]).

(2) A parameterized specification corresponding to the different tree traversal algorithms is the following:

PARAMETER DECLARATION: **data** =
  sorts: *data*
TARGET SPECIFICATION: **bintreetraversal(data)** =
  bintree(data) +
  sorts: *string*
  opns: PRE, IN, END: *bintree* → *string*
    EMPTY, LETTER, CONCAT, LADD, RADD (see Example 3.1)
  eqns: PRE(LEAF(D)) = LETTER(D)
    PRE(LEFT(D, T)) = PRE(RIGHT(D, T)) = LADD(D, PRE(T))
    PRE(BOTH(D, T, T')) = LADD(D, CONCAT(PRE(T), PRE(T')))

    IN(LEAF(D)) = LETTER(D)
    IN(LEFT(D, T)) = RADD(IN(T), D)
    IN(RIGHT(D, T)) = LADD(D, IN(T))
    IN(BOTH(D, T, T')) = CONCAT(IN(T), LADD(D, IN(T')))

    END(LEAF(D)) = LETTER(D)
    END(LEFT(D, T)) = END(RIGHT(D, T)) = RADD(END(T), D)
    END(BOTH(D, T, T')) = CONCAT(END(T), RADD(END(T'), D))

  eqns for CONCAT, LADD, RADD (see Example 3.1)

In **bintreetraversal(data)** we have specified three well-known tree traversal algorithms given by the operations PRE (preorder), IN (inorder) and END (endorder). The result of these algorithms is in each case a string of data. This requires that the part of the **string(param)** specification must be included in the target specification of **bintreetraversal(data)**.

4. Standard parameter passing

We now come to the problem of parameter passing. We need a mechanism which allows us to replace the formal parameters, given by the parameter declaration of a parameterized specification, by actual parameters, given by actual specifications. This mechanism will be called 'standard parameter passing'. The problem of 'parameterized parameter passing' where the actual parameters are parameterized specifications will be studied in Section 6.

The main problem for parameter passing is to develop suitable assignments, called 'parameter passing morphisms', from the formal to the actual parameters taking into account possible renamings and/or identifications of sorts and operations and also consistency of the actual parameter with respect to the parameter declaration.

Recall the parameterized specification **set0(data0)** of Fact 2.4 and consider **nat** as actual parameter. There is an 'obvious' morphism h: **data0** → **nat** which identifies the sort *data* with the sort *nat* in **nat**. It is not hard to see (intuitively) that this morphism h 'tells us' how we want to modify the parameterized type **set0(data0)** to get the desired data type **set0(nat)** with sorts *nat* and *set*, operations 0, SUCC, CREATE and INSERT, and with the intended two-sorted algebra A with A_{nat} = the natural numbers, and A_{set} = all finite sets of natural numbers, together with the desired operations on these carriers.

Now let's look at the same process but in a more abstract setting: let **para** = ⟨SPEC, SPEC1, T⟩ be a strongly persistent parameterized data type with SPEC = ⟨S, Σ, E⟩ and SPEC1 = SPEC + ⟨$S1$, $\Sigma 1$, $E1$⟩, and let **item** = ⟨SPEC', A'⟩ be a (non-parameterized) data type, where SPEC' = ⟨S', Σ', E'⟩. Then intuitively what we want for **para(item)** is some appropriate SPEC1'-algebra B' where SPEC1' = SPEC' + ⟨$S1'$, $\Sigma 1'$, $E1'$⟩, $S1' = S1$ and $\Sigma 1'$, $E1'$ are suitable reformulations of $\Sigma 1$, $E1$ respectively. The algebra B' depends, of course, on how we 'insert' A' in for the parameter of **para**. Again what we need is a means for assigning a sort in SPEC' for each sort in SPEC as well as an operation in SPEC' for each operation in SPEC. This process must be done carefully: it must extract from the SPEC'-algebra A', a SPEC-algebra A to which the functor T of **para** can be applied. This is accomplished by a pair of mappings ⟨h_S: $S \to S'$, h_Σ: $\Sigma \to \Sigma'$⟩ such that the resulting forgetful functor V_h: $Alg_{SPEC'} \to Alg_{SPEC}$ takes A' to SPEC-algebra A. Using the morphism h we are able to define the reformulations $\Sigma 1'$, $E1'$ of $\Sigma 1$, $E1$ mentioned above: $\Sigma 1' = h'(\Sigma 1)$ and $E1' = h'(E1)$ where $h' = (h'_S, h'_\Sigma)$ is defined by

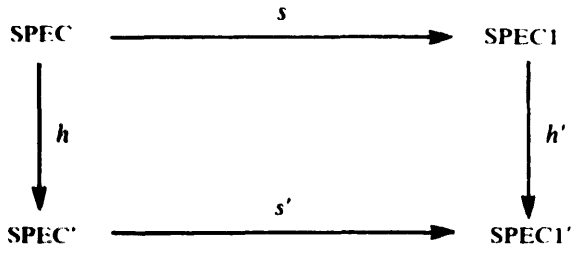
$$h'_S(s) = \text{if } s \in S1 \text{ then } s \text{ else } h_S(s)$$

and

$$h'_\Sigma(\sigma) = \text{if } \sigma \in \Sigma 1 \text{ then } \sigma \text{ else } h_\Sigma(\sigma).$$

The desired SPEC1'-algebra B' is then constructed by putting together the appropriate pieces of A' and $B = T(A)$. That is, for each $s \in S'$, $B'_s = A'_s$, and for each $s \in S1$, $B'_s = B_s$.

Note that the strong persistency of T together with the definition of A as $V_h(A')$ ensures that if $s \in S$, then $B_s = A_s = A'_{h(s)}$ so B'_s is well defined. In a similar manner we define the operations of B' from those of A' and B. However, there is another, rather neat, way to describe B' abstractly. Speaking informally (for now), the morphism h (given above) together with the 'inclusions' s: SPEC → SPEC1 and s': SPEC' → SPEC1' induces a similar morphism h' from SPEC1 to SPEC1' yielding a 'commuting diagram'



(we make this precise below). The morphisms s' and h' again induce forgetful functors $V_{s'}$ and $V_{h'}$ respectively. The algebra B' is characterized by the fact that $V_{s'}(B') = A'$ and $V_{h'}(B') = B = T(A) = T(V_h(A'))$.

To pull this together we must make it more precise. First we have to introduce the necessary morphism in a precise manner. This will allow us to give a precise statement of the parameter passing mechanism suggested by the above discussion.

4.1. Definition. A *specification morphism* $h: \langle S, \Sigma, E\rangle \to \langle S', \Sigma', E'\rangle$ consists of a mapping $h_S: S \to S'$ and an $(S^* \times S)$-indexed family of mappings, $h_\Sigma: \Sigma \to \Sigma'$ (where $h_{\Sigma(w,s)}: \Sigma_{w,s} \to \Sigma'_{h_S^*(w),h_S(s)}$). This data is subject to the condition that every axiom of E, when translated by h, belongs to E', shortly $h(E) \subseteq E'$. The morphism h is called *simple* if $S \subseteq S'$, $\Sigma \subseteq \Sigma'$, $E \subseteq E'$ and h_S, h_Σ are the inclusions.

Remark. In [5] we have used signature morphisms instead of specification morphisms as parameter passing morphisms. That means we now assume, in addition, that the translated equations of the parameter part belong to equations of the actual parameter. This is a simplification due to the basic algebraic case which is studied in this paper. Looking at our examples in Section 3, on one hand we have equations in the parameter declaration for fixed types like **nat** and **bool** which are intended to occur also in the actual parameter. On the other hand—and this is a drawback of the simplification—we may also have equations for operations like EQ in the parameter declaration such that EQ becomes an equivalence relation. Such equations must be expected to be present in all our actual parameters although EQ may be specified as equality on the actual parameter using different equations. This difficulty was avoided in [5] using signature morphisms and the concept of 'passing consistency'. But it can also be avoided for parameter declarations with requirements (see Section 8) where we will only translate the equations and not the requirements (see also [9] where theory morphisms are used).

4.2. Proposition. *If* $h: \text{SPEC} \to \text{SPEC}'$ *is a specification morphism, then there is a forgetful functor* $V_h: Alg_{\text{SPEC}'} \to Alg_{\text{SPEC}}$.

Proof. For $A' \in Alg_{\text{SPEC}'}$, $V_h(A') = A$ is given by

$$A_s = A'_{h(s)} \quad \text{for all } s \in S,$$

$$\sigma_A = h(\sigma)_{A'} \quad \text{for all } \sigma \in \Sigma,$$

where $\sigma_A: A_{h(s1)} \times \cdots \times A'_{h(sn)} \to A'_{h(s)}$ for $\sigma: s1 \cdots sn \to s$.

A becomes a SPEC'-algebra because we have $h(E) \subseteq E'$. $V_h(f')$ is defined by $V_h(f')_s = f'_{h(s)}$ for all $s \in S$. Since V_h preserves identities and composition, it is a functor. □

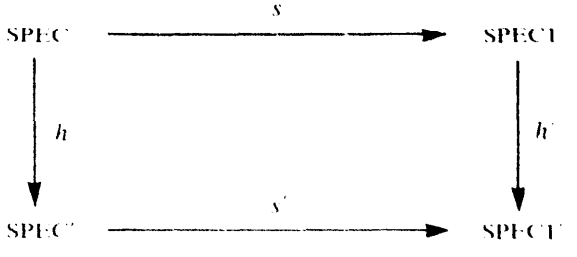
Remarks. (1) For each $\langle S', \Sigma' \rangle$-algebra A' we have that A' satisfies the translated axioms $h(E)$ iff $V_h(A')$ satisfies E.

(2) For all $A', B' \in Alg_{\text{SPEC}'}$ and each family $f' = (f'_s : A'_s \to B'_s)_{s \in S'}$ we have that f' is a $h(\Sigma)$-morphism iff $V_h(f')$ is a Σ-morphism.

4.3. Definition. Given a parameterized specification PSPEC = ⟨SPEC, SPEC1⟩ with SPEC1 = SPEC + ⟨S1, Σ1, E1⟩, a specification SPEC' = ⟨S', Σ', E'⟩ called *actual parameter specification*, and a specification morphism h: SPEC → SPEC', called *parameter passing morphism*, the mechanism of *standard parameter passing* is given by the following syntax, semantics, and semantical conditions:

Syntax:

The *syntax of standard parameter passing* is given by the following diagram, called *parameter passing diagram*:



where h is given as above, s and s' are simple specification morphisms and SPEC1', called *value specification*, is defined by

$$\text{SPEC1}' = \text{SPEC}' + \langle S1', \Sigma1', E1' \rangle$$

with

$$S1' = S1, \Sigma1' = h'(\Sigma1) \quad \text{and} \quad E1' = h'(E1)$$

where h': SPEC1 → SPEC1' is a specification morphism defined by

$$h'_S(x) = \text{if } x \in S1 \text{ then } x \text{ else } h_S(x)$$

and

$$h'_\Sigma(\sigma) = \text{if } (\sigma: w \to s) \in \Sigma1 \text{ then } \sigma: h^*_S(w) \to h_S(s) \text{ else } h_\Sigma(\sigma).$$

Notation. Instead of SPEC1 and SPEC1' we will often use the more intuitive notation SPEC1(SPEC), e.g., **set0(data0)**, and SPEC1(SPEC')$_h$ or simply SPEC1(SPEC') e.g., **set0(nat)**, respectively.

Semantics:

The *semantics of standard parameter passing* is given by

$$(F, T_{\text{SPEC}'}, T_{\text{SPEC1}'}) \quad \text{or} \quad T_{\text{SPEC1}'} \text{ for short}$$

where $F: Alg_{\text{SPEC}} \to Alg_{\text{SPEC1}}$ is the semantics of PSPEC (see Definition 2.3), $T_{\text{SPEC}'}$ and $T_{\text{SPEC1}'}$ are the initial algebras in $Alg_{\text{SPEC}'}$ and $Alg_{\text{SPEC1}'}$ respectively.

Semantical conditions:

The *semantical conditions for standard parameter passing* are the following:

(1) *actual parameter protection*, i.e. $V_{s'}(T_{\text{SPEC1}'}) = T_{\text{SPEC}'}$

(2) *passing computability*, i.e. $F(V_{h}(T_{\text{SPEC}'})) = V_{h'}(T_{\text{SPEC1}'})$.

Interpretation. The value specification SPEC1′ = SPEC1(SPEC′) is the result of replacing the formal parameter SPEC in SPEC1 = SPEC1(SPEC) by the actual parameter SPEC′. We use the notation SPEC1′ = SPEC1(SPEC′) to point out this replacement. But we have to keep in mind that $\Sigma 1$ and $E1$ are slightly changed to $\Sigma 1' = h'(\Sigma 1)$ and $E1' = h'(E1)$ respectively. This depends on the specific choice of the parameter passing morphism h: SPEC → SPEC′ which uniquely defines h'. Note that we do not have the semantical condition 'passing consistency', i.e., $V_h(T_{\text{SPEC}'}) \in Alg_{\text{SPEC}}$ which was assumed in [5]. Since our parameter passing morphism—unlike [5]—are specification morphisms, this condition is always satisfied, as stated in Remark (1) after Proposition 4.2. But we still need a similar nontrivial 'passing consistency' condition for h in the case of parameterized specifications with requirements (see Section 8): $V_h(T_{\text{SPEC}'})$ must satisfy the requirements of the parameter declaration.

The semantical condition 'actual parameter protection' means that the actual parameter SPEC′ is protected in the value specification SPEC1′. In other words, SPEC1′ is assumed to be an extension of SPEC′. Certainly this requirement meets an intuitive understanding of parameterized data types in software engineering. On the other hand, also slightly weaker requirements, like $V_{s'}(T_{\text{SPEC1}'}) \supseteq T_{\text{SPEC}'}$, may also meet the intuitive understanding. An interesting example might be adding of error elements for suitable sorts (see Remark 5.3). This is an example where the free construction F is not persistent. We cannot handle this case in general, at this point, because we also need the persistency of F to show (see Theorem 5.2) our second semantical condition: 'passing compatibility'. Passing compatibility means that the semantics of parameter passing, especially the transformation from $T_{\text{SPEC}'}$ to $T_{\text{SPEC1}'}$ is compatible with the semantics F of the given parameterized specification PSPEC.

We are convinced that passing compatibility in our sense—or in a slightly modified version—is an important requirement which pulls together the semantics of the formal and the actual parameter part. Unfortunately this or a similar condition is not considered for the procedure concept in CLEAR (see [7] and [8]) because the parameterized specifications are not assumed to be persistent. In [9], however, a similar feature is part of their concept of F-freeness. We will show (see Theorem

5.2) that persistency of the parameterized specification is necessary and sufficient for correctness of parameter passing. Moreover, passing compatibility is the key to showing induced correctness of the value specification in Theorem 5.4. Finally, let us point out that the syntax diagram for standard parameter passing becomes a pushout diagram in the category of specifications and specification morphisms which corresponds to the syntactical construction given by Ehrich [11]. Moreover, the semantical construction can be extended to a free functor $F' : Alg_{SPEC'} \to Alg_{SPEC'}$ which is persistent and satisfies $V_h F' = F V_h$ (see Extension Lemma 5.1).

4.4. Examples. (1) Given the parameterized specification **string(param)** (see Example 3.1) and the following actual parameter:

actual =
nat + **bool** +
opns: EQUAL : *nat nat* → *bool*
eqns: EQUAL(x, x) = TRUE
EQUAL(0, SUCC(x)) = EQUAL(SUCC(x), 0) = FALSE
EQUAL(SUCC(x), SUCC(y)) = EQUAL(x, y)

which is **nat** + **bool** enriched by an equality predicate on **nat**. Since we are interested in strings of natural numbers and natural numbers are already present in **param** (necessary for the operation LENGTH), we do not have to put another copy of **nat** into the actual parameter because our parameter passing morphisms are allowed to be noninjective (see below). If instead we are interested in strings of integers our actual parameter has to include **nat**, **bool** and **int** together with the equality, some equivalence or at least a reflexive relation EQR on **int**. In the latter case the parameter passing morphism h: **param** → **actual** would be injective mapping *data* to *int* and EQ to EQR. In our example above the strings of natural numbers the parameter passing morphism h: **param** → **actual** is defined by h(**nat**) = **nat**, h(**bool**) = **bool**, $h_S(data) = nat$ and h_Σ(EQ) = EQ'. Especially we have $h_S(nat) = h_S(data) = nat$ such that h_S is noninjective. Since h preserves the equations, it is a specification morphism. (This would have not been the case if we would have replaced the first equation by EQUAL(0, 0) = TRUE although this would be sufficient to specify the equality on **nat**.) Due to the construction in Definition 4.3 the value specification is the following:

string(actual) = **actual** +
sorts: *string*
opns: EMPTY : → *string*
LETTER : *nat* → *string*
CONCAT : *string string* → *string*
LADD, RADD, ROTATE, REVERSE, SHUFFLE, LENGTH
IS-EMPTY, IS-IN (like Example 3.1 with *data* replaced by *nat*)

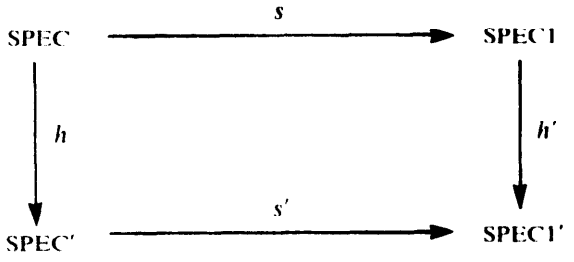
eqns: (like Example 3.1 with EQ replaced by EQUIV and variables D, D' of sort *data* replaced by variables N, M of sort *nat*).

(2) The value specification **string(actual)** can be used as actual parameter for the parameterized specification **bintree(data)** given in Example 3.3. There are, however, several choices for the parameter passing morphism h: **data** → **string(actual)**. The obvious one would be to define $h1_S(data) = string$ such that the value specification $\textbf{bintree(string(actual))}_{h1}$ defines binary trees with integer strings as labels in the nodes. But we can also define $h2_S(data) = nat$ such that the value specification $\textbf{bintree(string(actual))}_{h2}$ defines binary trees with natural numbers as labels in sort *bintree* and strings of natural numbers in sort *string*. Another possibility would be to define $h3_S(data) = bool$ which would lead to binary trees with boolean values as labels in the nodes. This example demonstrates that the value specification highly depends on the parameter passing morphism h which is reflected in the notation SPEC1(SPEC')_h but not in the usual notation without subscript h.

5. Correctness of standard parameter passing

In this section we will show that standard parameter passing is correct, i.e., the semantical conditions are satisfied, provided that the parameterized specification is persistent. Moreover persistency turns out to be necessary for actual parameter protection and hence for correctness. In the main result of this section we show that—roughly speaking—correctness of parameter passing implies correctness of the value specification. More precisely correctness of parameter passing, and correctness of the parameterized specification $\text{PSPEC} = \langle \text{SPEC}, \text{SPEC1} \rangle$ with respect to a parameterized persistent model data type $\text{PDAT} = \langle \text{MSPEC}, \text{MSPEC1}, T \rangle$ and correctness of the actual specification SPEC' with respect to an MSPEC'-algebra A implies correctness of the value specification SPEC1' with respect to the 'T-extension of A', provided that the model specifications MSPEC and MSPEC' are 'compatible' with the parameter passing morphism h: SPEC → SPEC', i.e., $h(\text{MSPEC}) \subseteq \text{MSPEC}'$. This induced correctness of the value specification is most important for correct design of software systems because once some basic specifications (e.g., **nat**, **int**, **bool**) and some basic parameterized specifications (e.g., **set0(data0)**, **string(param)**, **matrix(ring)**) are proven to be correct we have induced correctness for all possible value specifications (e.g., **set0(nat)**, **matrix(int)**, **string(matrix(int))**, **set0(string(matrix(int)))** etc.) The main tool to prove these correctness results is the following Extension Lemma which provides extensions of specifications and functors. A special case will be the Persistency Lemma which is stated separately for future reference.

5.1. Extension Lemma. (1) *Given a parameterized specification* $\text{PSPEC} = \langle \text{SPEC}, \text{SPEC1} \rangle$ *as in Definition 2.3 and a specification morphism* h: SPEC → SPEC', *then there is a well-defined parameter passing diagram,*



as given in Definition 4.3 *which is a pushout in the category of specifications and specification morphisms, i.e., we have*

(i) $s' \circ h = h' \circ s$, *and*

(ii) *for all specifications* SPEC'' *and all specification morphisms* $s'': \text{SPEC}' \to \text{SPEC}''$ *and* $h'': \text{SPEC1} \to \text{SPEC}''$ *satisfying* $s'' \circ h = h'' \circ s$ *there is a unique specification morphism* $f: \text{SPEC1}' \to \text{SPEC}''$ *such that*

$$f \circ s' = s'' \quad \text{and} \quad f \circ h' = h''.$$

(2) *Given a (strongly) persistent parameterized data type* $\text{PDAT} = \langle \text{PSPEC}, F \rangle$ *with* PSPEC *and a specification morphism h as above, then there is a (strongly) persistent functor* $F': Alg_{\text{SPEC}'} \to Alg_{\text{SPEC1}'}$, *called* ***extension of F via*** (h, s), *satisfying, for all* $A' \in Alg_{\text{SPEC}'}$,

$$V_h(F'(A')) = F(V_h(A')).$$

Moreover, $F'(A')$ *is uniquely determined by* A' *and* $B = F(V_h(A'))$ *in the following sense: For all* $B' \in Alg_{\text{SPEC1}'}$ *satisfying* $V_s(B') = A'$ *and* $V_h(B') = B$ *we already have* $B' = F'(A')$.

(3) *If in addition F is free (left adjoint to* V_s*), then F' is also free (left adjoint to* $V_{s'}$*).*

Corollary (Persistency Lemma). *Let* $\text{SPEC1} = \text{SPEC} + \langle S1, \Sigma 1, E1 \rangle$ *and* $\text{SPEC}' = \text{SPEC} + \langle S', \Sigma', E' \rangle$ *where S1 and S' as well as* $\Sigma 1$ *and* Σ' *are pairwise disjoint. Moreover, let*

$$\text{SPEC1}' = \text{SPEC1} + \langle S', \Sigma', E' \rangle = \text{SPEC}' + \langle S1, \Sigma 1, E1 \rangle$$

and assume that the free construction $F: Alg_{\text{SPEC}} \to Alg_{\text{SPEC1}}$ *is (strongly) persistent. Then also the free construction* $F': Alg_{\text{SPEC}'} \to Alg_{\text{SPEC1}'}$ *is (strongly) persistent.*

Proof of the Extension Lemma. (1) Defining SPEC1', s' and h' as in Definition 4.3, where h' becomes a specification morphism $h': \text{SPEC1} \to \text{SPEC1}'$, we have $s' \circ h = h' \circ s$. Note that s and s' are inclusions for sorts, operations and equations. Given specification morphisms s'' and h'' with $s'' \circ h = h'' \circ s$ we define $f: \text{SPEC1} \to \text{SPEC}''$ as follows:

$f_S(x) =$ if $x \in S1'$ then $h''_S(x)$ else $s''_S(x)$,

$f_\Sigma(y) =$ if $y \in \Sigma 1'$ then $h''_\Sigma(y)$ else $s''_\Sigma(y)$.

This implies $f \circ s' = s''$ and $f \circ h' = h''$ where the only nontrivial cases to consider are $x \in S$ where $f_S(h'_S(x)) = f_S(h_S(x)) = s''_S(h_S(x)) = h''_S(s_S(x)) = h''_S(x)$ and similarly for $y \in \Sigma$. The uniqueness of f follows from the fact that s' and h' are jointly surjective, i.e., each item in SPEC1' has a preimage under s' or h'. It remains to show that $f(E' + E1') \subseteq E''$. This follows from $s''(E') \subseteq E''$ and $f(E1') = f(h'(E1)) = h''(E1) \subseteq E''$ using the fact that s'' and h'' are specification morphisms.

(2) Given $A' \in Alg_{SPEC'}$, let $A = V_h(A') \in Alg_{SPEC}$ and $B = F(A) \in Alg_{SPEC1}$. We will construct a SPEC1'-algebra B' such that $V_{s'}(B') = A'$ and $V_{h'}(B') = B$. Then we will define $F'(A') = B'$ and we will define F' on SPEC'-morphisms such that F' becomes a strongly persistent functor $F' : Alg_{SPEC'} \to Alg_{SPEC1'}$ satisfying the desired property by construction of B'.

The following construction of B' essentially needs the (strong) persistency of F (i.e., $V_s(B) = A$) to be well defined: We assume strong persistency in the proof. The argument goes through in the general case.

$$B'_{s'} = \text{if } s \in S' \text{ then } A'_{s'} \text{ else } B_s \qquad \text{for all } s \in S' + S1',$$

$$\sigma_{B'} = \text{if } \sigma \in \Sigma' \text{ then } \sigma'_{A'} \text{ else } \sigma_B \qquad \text{for all } \sigma \in \Sigma' + \Sigma 1'.$$

This construction implies $V_{s'}(B') = A'$ and $V_{h'}(B') = B$ (see below) once we have shown that B' is a well-defined SPEC1'-algebra. For $\sigma: s1 \cdots sn \to sn+1$ we will show that $\sigma_{B'}$ is a function $\sigma_{B'} : B'_{s1} \times \cdots \times B'_{sn} \to B'_{sn+1}$. For $\sigma \in \Sigma'$ we have $\sigma_{B'} = \sigma_{A'}$ and $B'_{si} = A'_{si}$ for $i = 1, \ldots, n+1$ such that we are done. In the following we omit the subscripts S and Σ for h and h': For $\sigma \in \Sigma 1' = h'(\Sigma 1)$ we have $\sigma \in \Sigma 1$ with $\sigma: t1 \cdots tn \to tn+1$ for some $ti \in S + S1$ with $h'(ti) = si$ for $i = 1, \ldots, n+1$. Since $\sigma_B : B_{t1} \times \cdots \times B_{tn} \to B_{tn+1}$ we have to show that $B'_{si} = B_{ti}$. For each $i = 1, \ldots, n+1$ we have to consider the case $ti \in S$ and $ti \in S1$. For $ti \in S$ we have $si = h'(ti) = h(ti) \in S'$ such that

$$B'_{si} = A'_{si} = A'_{h(ti)} = V_h(A')_{ti} = A_{ti} = B_{ti}$$

because $V_s(B) = A$ by strong persistency of F. For $ti \in S1$ we have $si = h'(ti) = ti \in S1 = S1'$ such that

$$B'_{si} = B_{si} = B_{ti}.$$

This completes well-definedness of B'.

Next we will show $V_{h'}(B') = B$. By construction of B' this is clear for sorts $S1$ and operations $\Sigma 1$. It remains to be known for S and Σ which means $V_s(V_{h'}(B')) = V_s(B)$. Using again the persistency $V_s(B) = A$ and $V_{s'}(B') = A'$ we have

$$V_s(V_{h'}(B')) = V_h(V_{s'}(B')) = V_h(A') = A = V_s(B).$$

It remains to show that B' satisfies the equations E' and $E1'$. This follows from $V_{s'}(B') = A'$ and $V_{h'}(B') = B$ because A' satisfies E' with $s'(E') = E'$ and B satisfies $E1$ with $h'(E1) = E1'$ respectively (see Remark (1) after Proposition 4.2). Up to now we have shown that $F'(A') := B'$ is a SPEC1'-algebra satisfying the desired

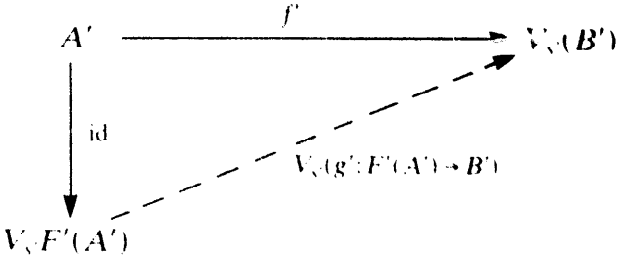
properties. To complete part (2) of the proof it remains to define F' on SPEC'-morphisms $f: A' \to A''$ such that $F'(f): F'(A') \to F'(A'')$ becomes a SPEC1'-morphism. Let

$$F'(f)_s = \text{if } s \in S' \text{ then } f_s \text{ else } F(V_h(f))_s$$

It is left to the reader to show that $F'(f)$ is well defined and, using Remark (2) after Proposition 4.2, to show that $F'(f)$ preserves $(\Sigma' + \Sigma 1)$-operations. Immediately from the definition it follows that F' preserves identities and composition of morphisms such that F' becomes a functor $F': Alg_{SPEC'} \to Alg_{SPEC1'}$.

Finally, to show the uniqueness property of $F'(A')$ let $B1' \in Alg_{SPEC1'}$ with $V_{s'}(B1') = A'$ and $V_{h'}(B1') = B$. Then we have $B1' = B' = F'(A')$ by construction of $B' = F'(A')$.

(3) We have to show that for each SPEC1'-algebra B' and each SPEC'-morphism $f': A' \to V_{s'}(B')$ there is a unique SPEC1'-morphism $g': F'(A') \to B'$ such that the following diagram commutes:



Let $A := V_h(A')$, $B := V_{h'}(B')$ and $f := V_h(f')$. Since F is left adjoint to V_s there is a unique SPEC1-morphism $g: F(A) \to B$ such that $V_s(g) = f$. The morphisms f' and g are combined to define $g': F'(A') \to B'$ by

$$g'_s = \text{if } s \in S' \text{ then } f'_s \text{ else } g_s$$

Using $V_{h'}(F'(A')) = F(A)$ and $V_{s'}(F'(A')) = A'$ (see step (2)), $V_s(B')_s = B'_s$ for $s \in S'$ and $B_s = B'_s$ for $s \in S1 = S1'$ we conclude that g'_s is in fact a mapping from $F'(A')_s$ to B'_s for all $s \in S' + S1$. By construction we have $V_{s'}(g') = f'$. In order to show uniqueness of g' let also g'' satisfy $V_{s'}(g'') = f'$. Then we have

$$V_s \circ V_{h'}(g'') = V_h \circ V_{s'}(g'') = V_h(f') = f$$

which implies $V_{h'}(g'') = g$ by uniqueness of g. Together with $V_{s'}(g'') = f'$ this implies $g' = g''$ by construction of g'.

It remains to show that g' is a $(\Sigma' + \Sigma 1')$-morphism. Since $V_{s'}(g') = f'$, it is clear that g' is a Σ'-morphism because f' is one. Using $g'' = g'$ we have $V_s \circ V_{h'}(g') = f$ which together with $V_s(g) = f$ implies $V_{h'}(g') = g$. Hence $V_{h'}(g')$ is a $(\Sigma + \Sigma 1)$-morphism which implies by the remark in Theorem 5.2 that g' is a $h'(\Sigma 1)$-morphism. By definition of $\Sigma 1'$ we have $\Sigma 1' = h'(\Sigma 1)$ such that g' is also a $\Sigma 1'$-morphism. □

5.2. Theorem (Correctness of standard parameter passing). *Given a parameterized specification* PSPEC = ⟨SPEC, SPEC1⟩, *then standard parameter passing is correct, i.e., for all actual parameters* SPEC′ *and all parameter passing morphisms* h: SPEC → SPEC′ *we have actual parameter protection and passing compatibility, iff* PSPEC *is persistent.* (*Without loss of generality we assume strong persistency in the proof.*)

Proof. First let us assume that PSPEC is persistent. We have to show actual parameter protection (Definition 4.3(1)) and passing compatibility (Definition 4.3(2)). Extension Lemma 5.1 can be used to apply the extended functor F' to the initial algebra T_{SPEC}. Lemma 5.1(3) implies that $F'(T_{SPEC'})$ is free and hence initial in $Alg_{SPEC1'}$, i.e., $T_{SPEC1'} = F'(T_{SPEC'})$. Strong persistency and extension property of F' (Extension Lemma 5.1(2)) means

$$V_{s'}(F'(T_{SPEC'})) = T_{SPEC'} \quad \text{and} \quad V_{h'}(F'(T_{SPEC'})) = F(V_h(T_{SPEC'}))$$

which together with $T_{SPEC1'} = F'(T_{SPEC'})$ implies actual parameter protection and passing compatibility respectively.

Conversely assume that the semantical conditions of Definition 4.3(1), (2) are satisfied for all actual parameters SPEC′ and all parameter passing morphisms h: SPEC → SPEC′. Further, let A be an arbitrary SPEC-algebra. Now let SPEC′ = SPEC + ⟨∅, ΣA, EA⟩ where ΣA contains a 0-ary operation symbol x for each $x \in A_s$ and $s \in S$ and EA consists of all those equations $t = t'$ with $t, t' \in T_{\Sigma+\Sigma A}$ which are true in A. This construction implies that $V_h(T_{SPEC'}) = A \in Alg_{SPEC}$ and actual parameter protection implies

$$\begin{aligned} V_s(F(A)) &= V_s(F(V_h(T_{SPEC'}))) = V_s(V_{h'}(T_{SPEC1'})) \\ &= V_h(V_{s'}(T_{SPEC1'})) = V_h(T_{SPEC'}) = A. \end{aligned}$$

Hence we have strong persistency of F and PSPEC. Note that SPEC′ becomes an 'infinite specification' in general. □

5.3. Remark (Passing compatibility). We have shown that persistency of PSPEC is necessary for correctness of parameter passing as given in Definition 4.3. It remains open how far persistency or a similar weaker condition is necessary for passing compatibility if actual parameter protection in the strong version of Definition 4.3(1) is not required. The following example shows that the nonpersistent parameterized specification PSPEC = ⟨SPEC, SPEC1⟩ 'insertion of an error element' with SPEC = ⟨{s}, ∅, ∅⟩ and SPEC1 = SPEC + ⟨∅, {e}, ∅⟩ is not passing compatible in general. Taking SPEC′ = **nat** + {SUCC(SUCC(x)) = SUCC(x)} and $h(s) = nat$ we have

$$T_{SPEC'} \cong \{0, 1\}, \quad F(V_h(T_{SPEC'})) \cong \{0, 1, e\} \quad \text{and} \quad T_{SPEC1'} \cong \{0, 1, e, f\}.$$

Hence we have no passing compatibility nor actual parameter protection because $F(V_h(T_{SPEC'}))$ and $V_{h'}(T_{SPEC1'})$ as well as $T_{SPEC'}$ and $T_{SPEC1'}$ have different cardinality.

5.4. Theorem (Induced correctness of value specifications). *Given*

(1) *a persistent parameterized specification* PSPEC = ⟨SPEC, SPEC1⟩ *which is correct with respect to a parameterized persistent model data type* PDAT = ⟨MSPEC, MSPEC1, T⟩,

(2) *an actual specification* SPEC′ *which is correct with respect to an* MSPEC′*-algebra* A',

(3) *a parameter passing morphism* h: SPEC → SPEC′, *and*

(4) *compatibility of the parameter passing morphism h with the model specifications* MSPEC *and* MSPEC′, *i.e.*,

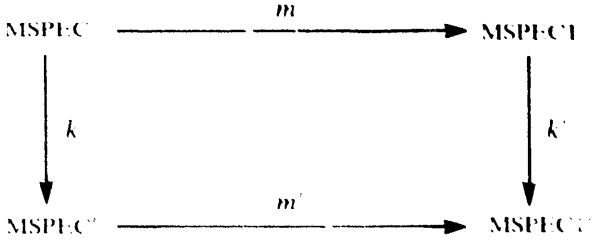
$$h(\text{MSPEC}) \subseteq \text{MSPEC}',$$

then we have

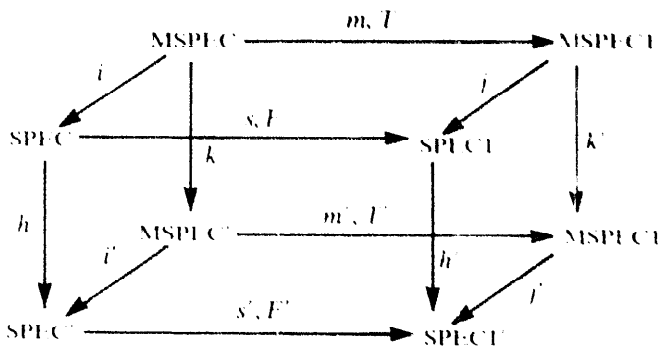
(5) *the value specification* SPEC1′ *is correct with respect to the* MSPEC1′ *algebra* $T'(A')$, *called* T-***extension of*** A', *which is uniquely defined by*

$$V_{m'}(T'(A')) = A' \quad \text{and} \quad V_{k'}(T'(A')) = T(V_k(A'))$$

where k: MSPEC → MSPEC′ *is the restriction of h* (*see* (4)) *and* MSPEC1′ *the model value specification in the model parameter passing diagram*



Proof. From assumptions (1)–(4) we are able to construct the following 3-cube of specifications and specification resp. parameter passing morphisms where the back square is defined as in (5), i' is inclusion from (2) the left square commutes by (4), and j' is the unique supplement in the right and bottom square which exists by the pushout properties of MSPEC1′ (see Extension Lemma 5.1(1)(ii)):



All morphisms are specification morphisms such that the Extension Lemma can be applied to yield a unique specification morphism j': MSPEC1′ → SPEC1′. The existence and uniqueness of the T-extension $T'(A')$ of A' follows from part (2) of the Extension Lemma applied to the algebra A'.

We want to show correctness of SPEC1′ with respect to $T'(A')$ which means $V_j'(T_{\text{SPEC1}'}) = T'(A')$ provided that we have $V_i'(T_{\text{SPEC}'}) = A'$ (assumption (2)). By the uniqueness property of $T'(A')$ (see Lemma 5.1(2)) it suffices to show that

$$V_m'(V_j(T_{\text{SPEC1}'})) = A' \quad \text{and} \quad V_k'(V_j'(T_{\text{SPEC1}'})) = T(V_k(A')).$$

Since SPEC′ is correct with respect to A' we have $V_i'(T_{\text{SPEC}'}) = A'$ and $V_s'(T_{\text{SPEC1}'}) = T_{\text{SPEC}'}$ by actual parameter protection (see Theorem 5.2) such that

$$V_m'(V_j'(T_{\text{SPEC1}'})) = V_i'(V_s'(T_{\text{SPEC1}'})) = V_i'(T_{\text{SPEC}'}) = A'.$$

On the other hand, passing compatibility, i.e., $V_h'(T_{\text{SPEC1}'}) = F(V_h(T_{\text{SPEC}'}))$, and correctness of PSPEC, i.e., $V_j \circ F = T \circ V_h$, implies

$$\begin{aligned} V_k'(V_j'(T_{\text{SPEC1}'})) &= V_j(V_h'(T_{\text{SPEC1}'})) = V_j(F(V_h(T_{\text{SPEC}'}))) \\ &= T(V_i(V_h(T_{\text{SPEC}'}))) = T(V_k(V_i'(T_{\text{SPEC}'}))) = T(V_k(A')). \end{aligned}$$

This completes the correctness proof. □

5.5. Example. We have the correctness of the parameterized specification **set0(data0)** with respect to the parameterized data type ⟨**data0**, **Mset0**, SET0⟩ shown in Theorem 2.4 and the correctness of **nat** with respect to the natural numbers $\mathbb{N}$. This implies, by Theorem 5.4, the correctness of the value specification **set0(nat)** in the introduction of Section 4 with respect to the **set0(nat)**-algebra $A = \text{SET0}'(\mathbb{N})$ with $A_{nat} = \mathbb{N}$, $A_{set} = \mathcal{P}_{\text{fin}}(\mathbb{N})$, 0_A and SUCC_A zero and successor in $\mathbb{N}$, and CREATE_A and INSERT_A defined as in Example 2.2 for $E = \mathbb{N}$.

6. Parameterized parameter passing

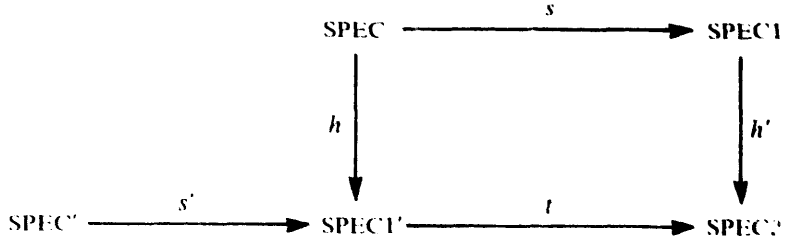
In this section we consider the parameter passing problem for parameterized actual parameters. In other words, we want to insert a parameterized specification, e.g., **matrix(ring)**, into another parameterized specification, e.g., **string(param)**, leading to a parameterized value specification, e.g., **string*matrix(ring)**. The parameterized value specification can be regarded as the composition of the given parameterized specifications. This composition, however, depends on the choice of the parameter passing morphism. The mechanism of parameterized parameter passing will be strictly defined analogous to that of standard parameter passing in Section 4. Actually the standard mechanism turns out to be a special case of the parameterized mechanism. In analogy to the correctness results for standard parameter passing in Section 5 we will show correctness of parameterized parameter passing and induced correctness of composite parameterized specifications.

6.1. Definition. Given parameterized specifications $\text{PSPEC} = \langle \text{SPEC}, \text{SPEC1} \rangle$ and $\text{PSPEC}' = \langle \text{SPEC}', \text{SPEC1}' \rangle$ with $\text{SPEC1} = \text{SPEC} + \langle S1, \Sigma 1, E1 \rangle$ and $\text{SPEC1}' = \text{SPEC}' + \langle S1', \Sigma 1', E1' \rangle$ respectively, where PSPEC′ is called *parameterized actual*

specification, and a specification morphism h: SPEC → SPEC1', called *parameter passing morphism*, the mechanism of *parameterized parameter passing* is given by the following syntax, semantics, and semantical conditions:

Syntax:

The syntax of *parameterized parameter passing* is given by the following diagram, called *parameterized parameter passing diagram*:



where h is given as above, s, s' and t are simple specification morphisms and PSPEC $*_h$ PSPEC' = ⟨SPEC', SPEC2'⟩, called *parameterized value specification* or *composite parameterized specification*, is defined by

$$\text{SPEC2}' = \text{SPEC1}' + \langle S2', \Sigma 2', E2' \rangle$$

with $S2' = S1$, $\Sigma 2' = h'(\Sigma 1)$, and $E2' = h'(E1)$ where h': SPEC1 → SPEC2' is a specification morphism defined by

$$h'_S(x) = \text{if } x \in S1 \text{ then } x \text{ else } h_S(x)$$

and

$$h'_\Sigma(\sigma) = \text{if } (\sigma: w \to s) \in \Sigma 1 \text{ then } \sigma: h'^*_S(w) \to h(s) \text{ else } h_\Sigma(\sigma).$$

Notation. Instead of SPEC1, SPEC1' and SPEC2' we will often use the more intuitive notation SPEC1(SPEC'), e.g., **string(param)**, SPEC1'(SPEC'), e.g., **matrix(ring)** and SPEC2'(SPEC') or SPEC1 $*_h$ SPEC1'(SPEC'), e.g., **string** $*_h$ **matrix(ring)** resp. We also use $*$ instead of $*_h$ if the parameter passing morphism is clear from the context.

Semantics:

The semantics of *parameterized parameter passing* is given by

$$(F, F', F *_h F') \quad \text{or} \quad F *_h F' \text{ for short,}$$

where F, F', and $F *_h F'$ are the semantics of PSPEC, PSPEC' and PSPEC $*_h$ PSPEC' respectively (see Definition 2.3).

Semantical conditions:

The *semantical conditions of parameterized parameter passing* are the following:

(1) *parameterized parameter protection*, i.e., for all $A' \in Alg_{\text{SPEC}'}$,

$$V_t(F *_h F'(A')) = F'(A').$$

(2) ***parameterized passing compatibility***, i.e., for all $A' \in Alg_{\text{SPEC}'}$,

$$V_{h'} \circ (F *_h F')(A') = F \circ V_h \circ F'(A').$$

Interpretation. Considering SPEC1' as an actual parameter of PSPEC in the sense of standard parameter passing (see Definition 4.3) SPEC2' is exactly the corresponding value specification. For parameterized parameter passing, however, it makes no sense to consider SPEC2' as a value specification because it still includes the formal parameter SPEC' of PSPEC'. Hence we consider the pair ⟨SPEC', SPEC2'⟩ to be the value of the parameterized parameter passing mechanism which motivates the name 'parameterized value specification'. On the other hand this pair corresponds to some sort of composition PSPEC' $*_h$ PSPEC'' of the given parameterized specifications PSPEC and PSPEC'' motivating the name 'composite parameterized specification'. Although the composition $*$ is actually a parameterized composition $*_h$, because it depends on the choice of the parameter passing morphism h, it makes sense to speak of a composition. Actually the composition $*$ behaves like the usual composition of functions where we have associativity, i.e., $f \circ (g \circ h) = (f \circ g) \circ h$, and compatibility with evaluation, i.e., $(f \circ g)(x) = f(g(x))$. The corresponding properties for $*$ will be shown in Section 7, e.g., we will have

string $*$ **matrix(int)** = **string(matrix(int))**.

The semantics $F *_h F'$ is also a parameterized composition. Note, that the usual composition $F \circ F'$ is not defined because the range of F' is $Alg_{\text{SPEC1'}}$ but the domain of F is Alg_{SPEC}. Hence we can only define the following composition $F \circ V_h \circ F'$ corresponding to the semantics of PSPEC and PSPEC'. This semantics, however, should be compatible with the semantics $F *_h F'$ of PSPEC $*_h$ PSPEC'. This compatibility is exactly the second semantical condition: parameterized passing compatibility.

Parameterized parameter protection means that the parameterized actual parameter $F'(A')$ is protected. If F and F' are persistent, then also $F *_h F'$ is persistent such that all actual parameters A' are protected by $F *_h F'$.

Finally let us point out that standard parameter passing is a strict special case of parameterized parameter passing (including syntax, semantics and semantical conditions) which will be shown in the following lemma.

6.2. Lemma. *If the parameterized actual specification* PSPEC' = ⟨SPEC', SPEC1'⟩ *is an actual specification, i.e.,* SPEC' = ∅, *then parameterized parameter passing coincides with standard parameter passing. Moreover, we have, for* SPEC' = ∅ *and* h: SPEC → SPEC1',

(1) SPEC1'(∅) = SPEC1', *and*

(2) SPEC1 $*_h$ SPEC1'(∅) = SPEC1(SPEC1'(∅))$_h$ = SPEC1(SPEC1')$_h$.

Proof. Replacing SPEC' in Definition 4.3 by SPEC1' in Definition 6.1 the syntactical construction of SPEC1' in Definition 4.3 is equal to that of SPEC2' in Definition 6.1.

For SPEC' = ∅ the free construction $F *_h F'$ in Definition 6.1 is given by $F *_h F'(T_\emptyset) = T_{SPEC2'}$ corresponding to the semantics of standard parameter passing. Moreover, F' is given by $F'(T_\emptyset) = T_{SPEC1'}$ such that the semantical requirements in the parameterized case are equivalent to those in the standard case. The formulas in (1) and (2) are well defined because SPEC' = ∅ implies that there is a unique parameter passing morphism $h'': \emptyset \to \emptyset$. Moreover, SPEC = SPEC' = ∅ in Definition 4.3 implies SPEC1' = SPEC1, i.e., the value specification applied to ∅ coincides with the given specification. This implies (1).

To show (2) we apply PSPEC * PSPEC' with SPEC' = ∅ to ∅ with $h'': \emptyset \to \emptyset$ which yields SPEC1 $*_h$ SPEC1'(∅) = SPEC2'. On the other hand, we have by (1) and coincidence of the syntactical constructions in Definitions 4.1 and 6.1 (shown above) SPEC1(SPEC1'(∅))$_h$ = SPEC1(SPEC1')$_h$ = SPEC2'. This proves (2). □

6.3. Theorem (Correctness of parameterized parameter passing). *Parameterized parameter passing is correct for persistent parameterized specifications. In more detail:*

Given (strongly) persistent parameterized specifications PSPEC = ⟨SPEC, SPEC1⟩ *and* PSPEC' = ⟨SPEC', SPEC1'⟩ *and a parameter passing morphism* h: SPEC → SPEC', *then we have*

(1) *parameterized parameter protection,*

(2) *parameterized passing compatibility,*

(3) *(strong) persistency of the composition* PSPEC $*_h$ PSPEC'.

Proof. Similar to the proof of Theorem 5.2 (correctness of standard parameter passing), Extension Lemma 5.1 can be applied to all algebras $F'(A') \in Alg_{SPEC1}$ for $A' \in Alg_{SPEC'}$ instead of $T_{SPEC'}$ showing properties (1) and (2) above. Note, that the property $F'(T_{SPEC'}) = T_{SPEC1'}$ in the proof of Theorem 5.2 can be replaced by $G(F'(A')) = F *_h F'(A')$ where G is the free construction $G: Alg_{SPEC1'} \to Alg_{SPEC2'}$. This follows from the well-known fact that the composition of free functors is free, i.e., $G \circ F' = F *_h F'$. It remains to show the persistency of $F *_h F'$. But this is clear because F' is persistent by assumption and G is persistent by Lemma 5.1(2). □

6.4. Theorem (Induced correctness of composite parameterized specifications). *Given*

(1) *a persistent parameterized specification* PSPEC = ⟨SPEC, SPEC1⟩ *which is correct with respect to a parameterized persistent model data type* PDAT = ⟨MSPEC, MSPEC1, T⟩,

(2) *a persistent parameterized actual specification* PSPEC' = ⟨SPEC', SPEC1'⟩ *which is correct with respect to a parameterized model data type* PDAT' = ⟨MSPEC', MSPEC1', T'⟩ *such that the forgetful functor* $V_1: Alg_{SPEC1'} \to Alg_{MSPEC1'}$ *is surjective,*

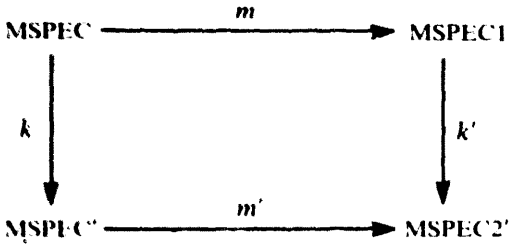
(3) *a parameter passing morphism* h: SPEC → SPEC1' *satisfying parameterized passing consistency, and*

(4) *compatibility with the model specifications* MSPEC *and* MSPEC1', *i.e.,* h(MSPEC) ⊆ MSPEC1', *then we have*

(5) *the composition* PSPEC $*_h$ PSPEC' *is correct with respect to the composite parameterized model data type*

$$\text{PDAT} *_k \text{PDAT}' := \langle \text{MSPEC}', \text{MSPEC2}', T *_k T' \rangle$$

where k: MSPEC $\to$ MSPEC1' *is the restriction of* h *(see* (4)), MSPEC2' *the model value specification in the model parameter passing diagram and the functor*



$T *_k T' : Alg_{\text{MSPEC}'} \to Alg_{\text{MSPEC2}'}$ *is uniquely defined on objects* $B' \in Alg_{\text{MSPEC}'}$ *by*

$$V_{m'}(T *_k T'(B')) = T'(B') \quad \textit{and} \quad V_k(T *_k T'(B')) = T(V_k(T'(B'))).$$

(6) *If, in addition, also* PDAT' *is persistent, then also the composition* PDAT $*_k$ PDAT' *is persistent.*

Proof. The proof of Theorem 5.4 can be extended using similar modifications as in the proof of Theorem 6.3 such that we obtain, for all $A' \in Alg_{\text{SPEC}'}$,

$$V_i(F *_h F'(A')) = G'(V_{i'}(F'(A'))) \qquad (\bigstar)$$

where $G' : Alg_{\text{MSPEC1}'} \to Alg_{\text{MSPEC2}'}$ is the extension of T via (k, m'). Surjectivity of V_i implies that $T *_k T'$ becomes a functor with $T *_k T'(B') = G'(T'(B'))$ for all $B' \in Alg_{\text{MSPEC}'}$.

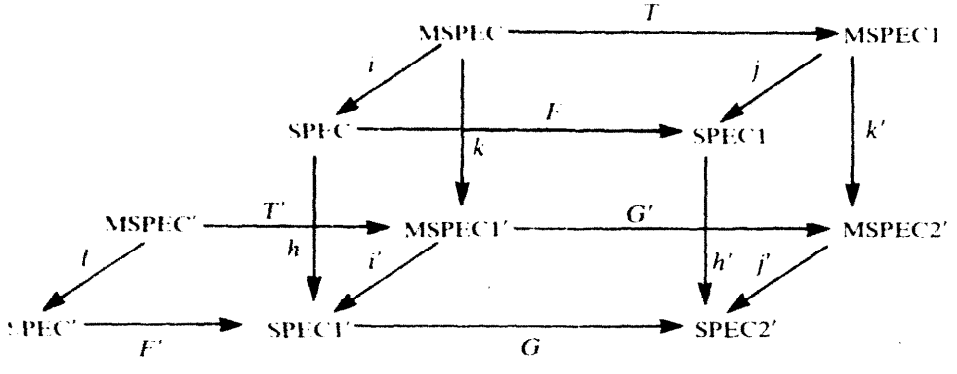
Correctness of PSPEC' means

$$V_i(F'(A')) = T'(V_i(A'))$$

such that, together with $(\bigstar)$, we obtain the desired correctness for all $A' \in Alg_{\text{SPEC}'}$,

$$V_i(F *_k F'(A')) = T *_k T'(V_i(A')).$$

The uniqueness of $T *_k T'(B')$ in (5) follows from the uniqueness of $T'(A')$ in Theorem 5.4(5).



Finally, persistency of G' implies persistency of $T *_k T'$ if T' is persistent. □

7. Iterated types and specifications

In this section we want to study the compatibility of standard and parameterized parameter passing. The aim is to build up large specifications like **bintree(string(matrix(int)))** from small basic specifications like **int**, **matrix(ring)**, **string(param)** and **bintree(data)**. Using the techniques of Sections 4 and 6 we are already able to build these specifications and we know that they are correct with respect to a canonical induced model. But we have not considered the problem of the extent to which the value specification is independent of the special construction we have chosen.

Perhaps the most obvious way to construct **bintree(string(matrix(int)))** is the following 'call by value' strategy where the actual specification **int** is inserted in **matrix(ring)**, the value specification **matrix(int)** is inserted in **string(param)** leading to **string(matrix(int))**, which finally is inserted into **bintree(data)**. This strategy uses only standard parameter passing. On the other hand, there is a 'call by name' strategy construction first **(bintree*string)(param)**, then **((bintree*string)*matrix)(ring)** by parameterized parameter passing, and finally **((bintree*string)*matrix)(int)** by standard parameter passing. Moreover, there is another 'call by name' strategy constructing first **(string*matrix)(ring)**, then **(bintree*(string*matrix))(ring)** and finally **(bintree*(string*matrix))(int)**. Last but not least, there are two 'mixed strategies' leading to **(bintree*string)(matrix(int))** and **bintree((string*matrix)(int))** respectively.

We will show that all these strategies are leading to the same value specification. This corresponds to a Church–Rosser-property of standard and parameterized parameter passing.

The key to show this result is to show that the composition of parameterized specifications is associative like the composition of functions, e.g., $(f \circ g) \circ h = f \circ (g \circ h)$, and that composition of parameterized specifications is compatible with standard actualization like composition and evaluation of functions, e.g., $(f \circ g)(x) = f(g(x))$. Hence we can apply the usual rules known for the evaluation of composite functions. This is remarkable because composition and standard actualization of parameterized data types depend on the choice of the parameter passing morphisms. Actually in the composition and the compatibility results it is shown that there are always canonical induced parameter passing morphisms such that all the parameter passing mechanisms are correct provided that all parameterized specifications are persistent.

7.1. Theorem (Associativity of composition) (see [11]). *The composition of persistent parameterized specifications is associative. In more detail we have: Given persistent*

parameterized specifications

$$\text{PSPEC}i = \langle \text{SPEC}(i-1), \text{SPEC}i \rangle \quad \textit{for } i = 1, 3, 5$$

and parameter passing morphisms

$$h0 : \text{SPEC0} \to \text{SPEC3} \quad \textit{and} \quad h2 : \text{SPEC2} \to \text{SPEC5},$$

then there is a canonical parameter passing morphism $h2 * h0 : \text{SPEC0} \to \text{SPEC7}$ ***such that we have***

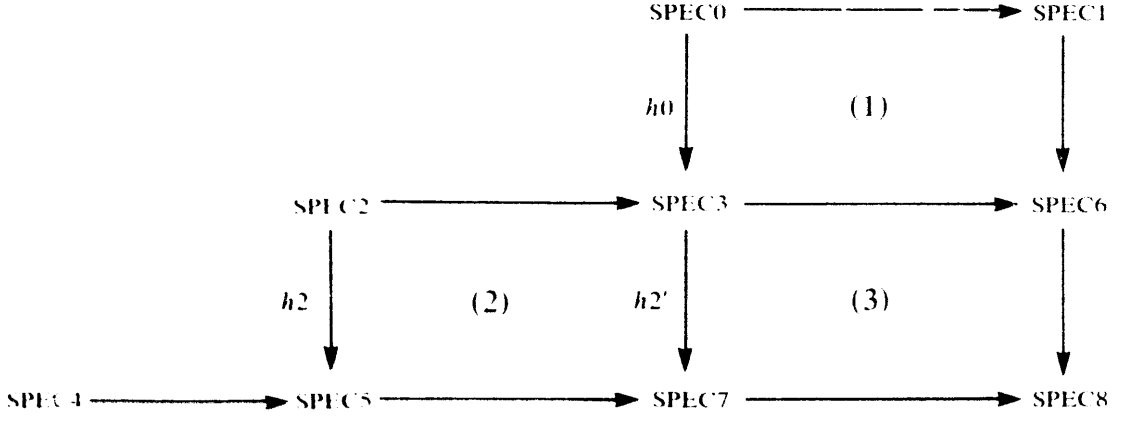
$$(\text{PSPEC1} *_{h0} \text{PSPEC3}) *_{h2} \text{PSPEC5} = \text{PSPEC1} *_{h2 * h0} (\text{PSPEC3} *_{h2} \text{PSPEC5})$$

where all compositions are well-defined and SPEC7 ***is the value specification of*** $\text{PSPEC3} *_{h2} \text{PSPEC5}$.

Remark. Using the notation SPEC1(SPEC0) for PSPEC1, SPEC1 * SPEC3(SPEC2) for PSPEC1 * PSPEC3, and similarly for the other parameterized specifications, we obtain the following equivalent formulation:

$$((\text{SPEC1} *_{h0} \text{SPEC3}) *_{h2} \text{SPEC5})(\text{SPEC4}) = (\text{SPEC1} *_{h2 * h0} (\text{SPEC3} *_{h2} \text{SPEC5}))(\text{SPEC4})$$

where both specifications are equal to SPEC8, and SPEC6 is the value specification of $\text{PSPEC1} *_{h0} \text{PSPEC3}$ in the following parameter passing diagrams (1)–(3):



The canonical parameter passing morphism $h2 * h0$ is defined by

$$h2 * h0 := h2' \circ h0 : \text{SPEC0} \to \text{SPEC7}$$

where $h2'$ is induced from $h2$ (see Definition 6.1).

Proof of Theorem 7.1. The main idea of the proof is that diagrams (1)–(3) above are pushouts in the category of specifications and specification morphisms (see Extension Lemma 5.1) which can be combined in different ways.

It is well known from category theory (see [6, 28, 36]) that the composition of pushouts is again a pushout, i.e., with diagrams (1) and (3) also (1)+(3), and with

(2) and (3) also (2)+(3) is a pushout. Combining one way we have

$\text{SPEC6} = \text{SPEC1} *_{h0} \text{SPEC3}(\text{SPEC2})$ (1) is a pushout,

$\text{SPEC8} = \text{SPEC6} *_{h2} \text{SPEC5}(\text{SPEC4})$ (2)+(3) is a pushout,

and combining the other way we have

$\text{SPEC7} = (\text{SPEC3} *_{h2} \text{SPEC5})(\text{SPEC4})$ (2) is a pushout,

$\text{SPEC8} = (\text{SPEC1} *_{h2' \circ h0} \text{SPEC7})(\text{SPEC4})$ (1)+(3) is a pushout.

Hence we obtain, by substitution,

$$((\text{SPEC1} *_{h0} \text{SPEC3}) *_{h2} \text{SPEC5})(\text{SPEC4}) = (\text{SPEC1} *_{h2*h0} (\text{SPEC3} \cdot \quad \text{SPEC5}))(\text{SPEC4})$$

which is (see the remark) the desired result. □

7.2. Corollary (Compatibility of composition and actualization). *Standard actualization of composite parameterized specifications is equal to iterated standard actualization of parameterized specifications. In more detail we have: Given persistent parameterized specifications*

$\text{PSPEC1} = \langle \text{SPEC0}, \text{SPEC1} \rangle$ *and* $\text{PSPEC3} = \langle \text{SPEC2}, \text{SPEC3} \rangle$

an actual specification SPEC5 *and parameter passing morphisms*

$h0 : \text{SPEC0} \to \text{SPEC3}$ *and* $h2 : \text{SPEC2} \to \text{SPEC5}$.

Then there is a canonical parameter passing morphism $h2 * h0 : \text{SPEC0} \to \text{SPEC7}$ *such that we have*

$$(\text{SPEC1} *_{h0} \text{SPEC3})(\text{SPEC5})_{h2} = \text{SPEC1}(\text{SPEC3}(\text{SPEC5})_{h2})_{h2*h0}$$

where $h2 * h0$ *is defined as in Theorem* 7.1.

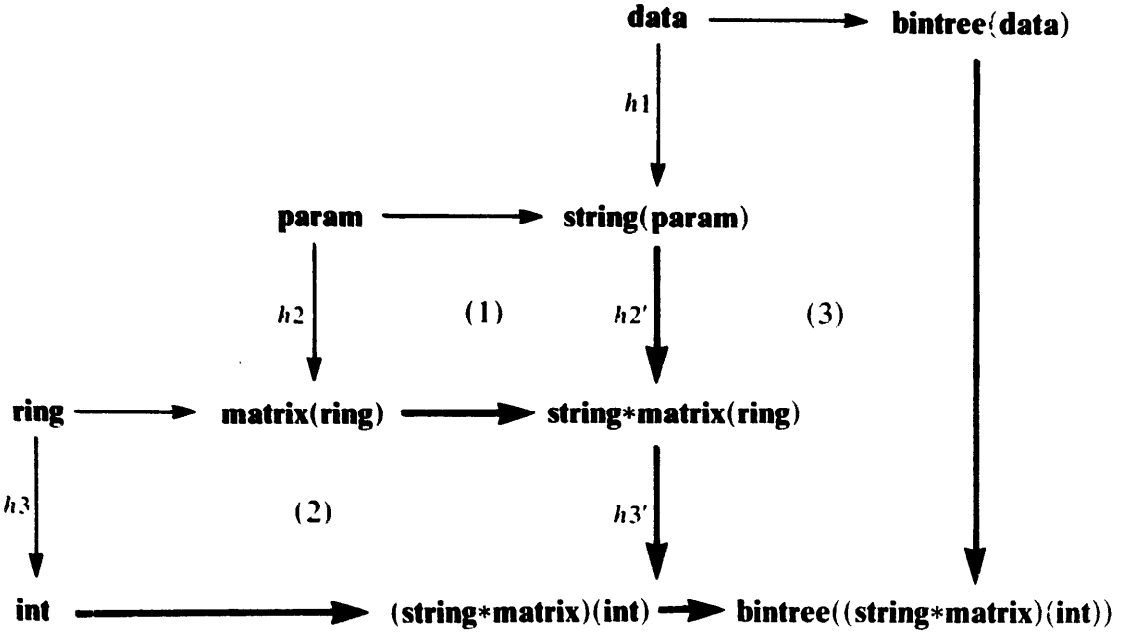
Proof. Take $\text{SPEC4} = \emptyset$ in Theorem 7.1 and use Lemma 6.2:

$$\begin{aligned}
&(\text{SPEC1} *_{h0} \text{SPEC3})(\text{SPEC5})_{h2} \\
&= ((\text{SPEC1} *_{h0} \text{SPEC3}) *_{h2} \text{SPEC5})(\emptyset) && \text{(by Lemma 6.2)} \\
&= \text{SPEC1} *_{h2*h0} (\text{SPEC3} *_{h2} \text{SPEC5})(\emptyset) && \text{(by Theorem 7.1)} \\
&= \text{SPEC1}(\text{SPEC3} *_{h2} \text{SPEC5}(\emptyset))_{h2*h0} && \text{(by Definition 6.1)} \\
&= \text{SPEC1}(\text{SPEC3}(\text{SPEC5})_{h2})_{h2*h0} && \text{(by Definition 6.1).}
\end{aligned}$$

□

7.3. Remark (Iterated parameter passing). The associativity of the composition of parameterized specifications (Theorem 7.1) and the compatibility of composition and standard actualization (Theorem 7.2) are the basic results to show that the result of iterated parameter passing is independent of the 'evaluation strategy' in which parameterized and standard parameter passing are applied. Instead of a formal treatment of this phenomenon let us consider the following example.

Assume that we have given the parameterized specifications **bintree(data)**, **string(param)**, **matrix(ring)**, the actual specification **int** and parameter passing morphisms $h1$, $h2$ and $h3$ which are passing consistent, i.e., we have given an 'iterated parameter passing situation' defined by the slim arrows in the following diagram:



An evaluation strategy consists of an arbitrary sequence of the following two steps until no more steps can be applied:

Parameterized step. Apply parameterized parameter passing to one triangle (except the left) and consider the new iterated parameter passing situation resp. value specification.

Standard step. Apply standard parameter passing to the left most triangle and consider the new iterated parameter passing situation resp. value specification.

The result of iterated parameter passing with respect to a given evaluation strategy is the resulting value specification.

In our example we first have applied a parameterized step (1) and then two standard steps (2) and (3). Note that the parameter passing morphism for (3) is $h3' \circ h2' \circ h1$ where $h2'$ and $h3'$ are the induced from $h2$ and $h3$ respectively. This is a 'mixed strategy'. The 'call by value' strategy consists of standard steps only applied from left to right leading to the value specification **bintree(string(matrix(int)))**. The 'call by name' strategy consists of parameterized steps applied from right to left followed by one standard step such that we obtain the value specification **((bintree * string) * matrix)(int)**.

7.4. Corollary (Independence of strategies). *The result of iterated parameter passing is independent of the choice of the evaluation strategy applied to a given iterated parameter passing situation.*

Proof. The proof follows by iterated application of Theorems 7.1 and 7.2. □

8. Conclusion

Let us start with a short summary of the main constructions and results in this paper.

A parameterized specification PSPEC = ⟨SPEC, SPEC1⟩ consists of a pair of specifications where the parameter declaration SPEC is included in the target specification SPEC1. Parameter passing from the formal parameter SPEC to an actual parameter SPEC' is given by a 'specification morphism' f: SPEC → SPEC'. The value specification SPEC1' is more or less a 'renaming' of the SPEC-parts of SPEC1 by the corresponding SPEC'-parts of the actual parameter. Mathematically, SPEC1' is the pushout object of SPEC1 and SPEC' via f in the category CATSPEC of algebraic specifications and specification morphisms. Moreover, we can also pass parameterized specifications as actual parameters leading to parameterized value specifications. The first important result is correctness of parameter passing (see Theorems 5.2 and 6.3) meaning that the semantical conditions 'parameter protection' and 'passing compatibility' (see Definitions 4.3 and 6.1) are satisfied if the parameterized specifications are persistent. The benefit of correct parameter passing is not only economy in presentation but we also have automatically induced correctness of all the value specifications provided that the parameterized specification and all the actual specifications are correct (see Theorems 5.3 and 6.4). This is a most important property in order to build up larger data types and software systems from small pieces in a correct way. Similar to procedures in programming languages parameterized specifications promise to become one of the most important structuring principle for the design of software systems. The third important result is based on the associativity of the composition of parameterized specifications (see Theorem 7.1): We are able to show that all different evaluation strategies—including 'call by name' and 'call by value'—for iterated parameter passing situations lead to the same result (see Corollary 7.4). Technically all the results are based on Extension Lemma 5.1 which allows to extend specifications and also functors.

The theory in this paper is called 'the basic algebraic case' because it is the common basis for several other approaches like those in [5, 13, 21, 22, 27, 32].

It is also closely related to the concept of parameterized specifications in [37, 29] and to the procedures in CLEAR (see [9]) where, however, the semantics is not functorial. It seems possible to study semantical conditions like 'actual parameter protection' and 'passing compatibility' in these more general frameworks allowing in addition to algebraic equations 'initial restrictions' and 'constraints' respectively (see below).

Although our theory in the basic algebraic case seems to be very smooth, it turns out that the applicability to common data types in software practice is somewhat limited. In several applications we need an equality predicate on the formal para-

meter, like EQ: *data data* → *bool* in the parameterized specification **set(data)**. In order to show correctness of such specifications we need requirements for the operation EQ making sure that EQ is really an equality predicate on all admissible parameter algebras A, i.e., $EQ_A(d, d') = \textit{if } d = d' \textit{ then}$ TRUE *else* FALSE for all $d, d' \in A_{data}$. Unfortunately there seem to be no equations but only negative conditional axioms to assure this property, e.g., $EQ(X, X) =$ TRUE and $X \neq Y \Rightarrow EQ(X, Y) =$ FALSE (see [4]). Moreover, we have to make sure that the **bool**-part of A exactly consists of two distinct elements TRUE and FALSE. The most convenient way to obtain these properties is the requirement 'initial(**bool**)' which makes sure that the **bool**-part of A is isomorphic to the initial boolean algebra $T_{\mathbf{bool}}$.

Such requirements are called 'initial restrictions' in [37] and 'constraints' in [9] which become special cases of our more general notion of requirement. In [13] a set R is called 'set of requirements' on a specification SPEC if for all $r \in R$ there is a well-defined subclass VALID(r) of all SPEC-algebras. This very general definition is easy to handle and also allows to state all kinds of predicate formulas as requirements, especially the negative conditional axioms for EQ as given above. Hence a parameterized specification of **set(data)** with requirements can be given as in Example 2.5 where, however, the equation $EQ(X, X) =$ TRUE in the parameter declaration is replaced by the following:

REQUIREMENTS:
initial(**bool**)
$EQ(X, X) =$ TRUE
$X \neq Y \Rightarrow EQ(X, Y) =$ FALSE.

This parameterized specification **set(data)** with requirements is similar to that in [4] except of different requirement handling which has significant consequences for the semantics.

Another important feature of parameterized specification with requirements is the possibility to specify bounded data types such as arrays of fixed bounded length B. The bound B, however, is supposed to be a parameter which may take different values in different actual parameters. In most cases it seems to be convenient to construct the specification of bounded types as an extension of bounded natural numbers **nat(bound)** where **nat** is some correct specification of natural numbers with operations 0, SUCC and ADD.

8.1. Example (nat(bound))

PARAMETER DECLARATION: **bound** =
nat +
opns: BOUND: → *nat*
REQUIREMENTS:
initial(**nat**)
TARGET SPECIFICATION: **nat(bound)** =
bound +

sorts: *bnat*
opns: MOD: *nat* → *bnat*
MODO: → *bnat*
MODSUCC: *bnat* → *bnat*
MODADD: *bnat bnat* → *bnat*
eqns: MOD(ADD(BOUND, *n*)) = MOD(*n*)
MODO = MOD(0)
MODSUCC(MOD(*n*)) = MOD(SUCC(*n*))
MODADD(MOD(*n*1), MOD(*n*2)) = MOD(ADD(*n*1, *n*2)).

Remark. Note that the requirement initial(**nat**) implies that BOUND picks out some well-defined natural number *B*. Without the initiality requirement, BOUND may define a new value which does not correspond to any natural number. The semantics of our specification is $\mathbb{N}$ mod *B* (natural numbers modulo *B*).

In [13] it is shown that all the results of this paper can be extended to the case with requirements. The main idea is to prove an *R*-Extension Lemma which generalizes Extension Lemma 5.1 of this paper to the case of parameterized specifications with requirements *R* and 'passing consistent' morphisms. Up to now we have only considered the initial algebra approach where the semantics of parameterized types is given by free constructions. In [32] it is shown how to make use of inductively specified operations and in [21, 22] how to extend the basic algebraic case from initial to final algebra semantics: The main idea in [21, 22] is to replace the free construction $F: Alg_{SPEC} \to Alg_{SPEC'}$ by a quotient functor $CF: Alg_{SPEC} \to Alg_{SPEC'}$ of F such that persistency of F is equivalent to persistency of CF. Ganzinger [22, Theorem 1] shows that if CF is persistent, then it is the right adjoint (cofree functor) of the forgetful functor, provided we take the subcategory of SPEC'-algebras B that are generated by their parameter part as the range of CF. Moreover, Ganzinger [22, Theorem 5] shows that the extension CF' of a persistent cofree functor CF along a passing consistent parameter passing morphism is again a persistent cofree functor. This allows to replace in all those definitions, constructions and results of Sections 2 to 7 where persistency is assumed the free construction F by the cofree construction CF. Hence we obtain a final algebraic theory for parameterized specifications. Similar problems are studied in [27] for inequalities which is a special case of requirements. In [39] it is suggested to study also other semantics than the initial and final case. This might be extended to parameterized specifications provided that part (3) of the Extension Lemma remains valid.

Another important issue is the implementation of parameterized data types extending the algebraic implementation concept in [18]. A first more or less syntactical treatment is given in [22] where, however, a slightly different implementation concept is used. Abstract implementation and parameter substitution based on initially restricting algebraic theories (see [37]) are studied in [29].

An interesting application of parameter passing with requirements is the identification of common subtypes in different specifications.

In CLEAR (see [9]) identification of common subtypes is handled using general colimit constructions. This problem should become a special case of parameter passing such that no additional feature in syntax and semantics is needed.

Finally, let us note that the algebraic concept of parameterized specifications is already used in the programming language MODLISP (see [30]) which is used for the implementation of the algebraic manipulation system NEWSPAD (see [31]). NEWSPAD is an ambitious endeavor to structure a computer algebra system based on categories and functors, with user defined parameterized types (like **matrix(ring)** as in Example 3.2) playing a central role. Although their current work deals with models only (functions and representations are always given) one would hope that this work would employ specification in a more central way in future development.

Acknowledgment

This paper is part of a common project of the ADJ-group at IBM Yorktown Heights and the ADJ-group at the Technische Universität Berlin. Thanks to all of them especially to W. Fey and P. Padawitz and in addition to R. Burstall, H.-D. Ehrich and J.A. Goguen and to the organizer and participants of the Aarhus Workshop on Program Specification for several fruitful discussions on the subject of this paper.

Last but not least, the authors are grateful to the referees for many thoughtful and helpful suggestions.

References

[1] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright (ADJ-Group), Abstract data types as initial algebras and correctness of data representations, *Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structure* (1975) pp. 89–93.

[2] J.W. Thatcher, E.G. Wagner and J.B. Wright (ADJ-Group), Specification of abstract data types using conditional axioms, IBM Res. Rept. RC-6214, 1976.

[3] J.A. Goguen, J.W. Thatcher and E.G. Wagner (ADJ-Group), An initial algebra approach to the specification, correctness, and implementation of abstract data types, IBM Res. Rept. RC-6487, 1976; in: R.T. Yeh, Ed., *Current Trends in Programming Methodology, IV: Data Structuring* (Prentice-Hall, Englewood Cliffs, NJ, 1978) pp. 80–149.

[4] J.W. Thatcher, E.G. Wagner and J.B. Wright (ADJ-Group), Data type specification: Parameterization and the power of specification techniques, *Proc. SIGACT 10th Annual Symp. on Theory of Computing* (1978) pp. 119–132.

[5] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E.G. Wagner and J.B. Wright (ADJ-Group), Parameterized data types in algebraic specification languages, *Proc. 7th ICALP*, Noordwijkerhout, 1980; Lecture Notes in Comput. Sci. **85** (Springer, Berlin, 1980) pp. 157–168 (short version).

[6] M.A. Arbib and E.G. Manes, *Arrows, Structures and Functors: The Categorical Imperative* (Academic Press, New York, 1975).

[7] R.M. Burstall and J.A. Goguen, Putting theories together to make specifications, *Proc. 1977 IJCAI* (MIT, Cambridge, MA, 1977).

[8] R.M. Burstall and J.A. Goguen, Semantics of CLEAR, Working Note (draft version), Dept. of Artificial Intelligence, Edinburgh University, 1979.

[9] R.M. Burstall and J.A. Goguen, The semantics of CLEAR, a specification language, *Proc. 1979 Copenhagen Winter School on Abstract Software Specifications*, Lecture Notes in Comput. Sci. **86** (Springer, Berlin, 1980) pp. 292–332.

[10] P.M. Cohn, *Universal Algebra* (Harper & Row, New York, 1965).

[11] H.-D. Ehrich, On the theory of specification, implementation and parameterization of abstract data types, Res. Rept., Univ. of Dortmund, 1978.

[12] H.-D. Ehrich and V.G. Lohberger, Constructing specifications of abstract data types by replacements, *Proc. Internat. Workshop on Graph Grammars and Appl. Comput. Sci. and Biology*, Bad Honnef, Lecture Notes in Comput. Sci. **73** (Springer, Berlin, 1979) pp. 180–191.

[13] H. Ehrig, Algebraic theory of parameterized specifications with requirements, *Proc. 6th CAAP*, Genova, Lecture Notes in Comput. Sci. **112** (Springer, Berlin, 1981) pp. 1–24.

[14] H. Ehrig and W. Fey, Methodology for the specification of software systems: From requirement specifications to algebraic design specifications, *Proc. GI 81*, München.

[15] H. Ehrig and H.-J. Kreowski, Kategorien und Funktoren, LV-Skript SS 1980, Fachbereich 20, Techn. Univ. Berlin, 1980.

[16] H. Ehrig, H.-J. Kreowski and P. Padawitz, Some remarks concerning correct specification and implementation of abstract data types, Rept. 77-13, Techn. Univ. Berlin, 1977.

[17] H. Ehrig, H.-J. Kreowski and P. Padawitz, Stepwise specification and implementation of abstract data types, Rept., Tech. Univ. Berlin, 1977; in: *Proc. 5th ICALP*, Udine, Lecture Notes in Comput. Sci. **62** (Springer, Berlin, 1978) pp. 205–226.

[18] H. Ehrig, H.-J. Kreowski and P. Padawitz, Algebraic implementation of abstract data types: Concept syntax, semantics, correctness, *Proc. 7th ICALP*, Noordwijkerhout, Lecture Notes in Comput. Sci. **85** (Springer, Berlin, 1980) pp. 142–156 (extended and revised version in: *Theoret. Comput. Sci.* **20** (3) (1982) 209–263, with B. Mahr).

[19] W. Fey, Some examples of algebraic specifications and implementations Part 4, Rept. No. 82-2, Tech. Univ. Berlin, 1982.

[20] C. Floyd, Software Engineering—Entwurf und Spezifikation, *Proc. 2nd German Chapter of The ACM-Meeting* (Teubner, 1981).

[21] H. Ganzinger, Parameterized specifications: Parameter passing and optimizing implementation, *TOPLAS*, to appear.

[22] H. Ganzinger, A final algebra semantics for parameterized specification (draft version), UC Berkeley, 1980.

[23] J.A. Goguen and J. Tardo, OBJ-0 preliminary users manual, UCLA, Los Angeles, CA, 1977.

[24] G. Graetzer, *Universal Algebra* (Van Nostrand, Princeton, NJ, 1968).

[25] J.V. Guttag, The specification and application to programming of abstract data types, Tech. Rept CSRG-59, Univ. of Toronto, Comput. Systems Research Group, 1975.

[26] J.V. Guttag, Abstract data types and the development of data structures, *Proc. Conf. on Data Abstraction, Definition, and Structure*; *SIGPLAN Notices* **8** (1976).

[27] G. Hornung and P. Raulefs, Initial and terminal algebra semantics of parameterized abstract data type specification with inequalities, *Proc. 6th CAAP*, Genova, 1981; Lecture Notes in Comput. Sci. **112** (Springer, Berlin, 1981) pp. 224–237.

[28] H. Herrlich and G. Strecker, *Category Theory* (Allyn & Bacon, Rockleigh, 1973).

[29] U.L. Hupbach, Abstract implementation and parameter substitution, *Proc. 3rd Hungarian Comput. Sci. Conf.*, Budapest (1981).

[30] R.D. Jenks, *MODLISP: An Introduction*, Lecture Notes in Comput. Sci. **72** (Springer, Berlin, 1979) pp. 466–480.

[31] R.D. Jenks and M.B. Trager, A language for computer algebra, *Proc. 1981 ACM Symp. on Symbolic and Algebraic Computation*, 1981.

[32] H.A. Klaeren, On parameterized abstract software modules using inductively specified operations, Res. Rept. Nr. 66, Technische Hochschule Aachen, 1980.

[33] H.-J. Kreowski, Algebra für Informatiker, LV-Skript WS 78/79, Fachbereich 20, Tech. Univ. Berlin, 1978.

[34] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, Abstraction mechanisms in CLU, *Comm. ACM* **20** (8) (1977) 564–576.

[35] B. Liskov and S. Zilles, Programming with abstract data types, *SIGPLAN Notices* **9** (4) (1977) 50–59.

[36] S. MacLane, *Categories for the Working Mathematician* (Springer, New York/Heidelberg/Berlin, 1971).

[37] H. Reichel, Initially restricting algebraic theories, *Proc. MFCS'80*, Rydzyna, 1980; Lecture Notes in Comput. Sci. **88** (Springer, Berlin, 1980) pp. 504–514.

[38] D. Scott, Mathematical concepts in programming language semantics, *Proc. AFIPS Spring Joint Comp. Conf.* (1962) pp. 225–232.

[39] M. Wirsing and M. Broy, Abstract data types as lattices of finitely generated models, *Proc. MFCS'80*, Rydzyna, 1980; Lecture Notes in Comput. Sci. **88** (Springer, Berlin, 1980) pp. 673–685.

[40] W.A. Wulf, R.L. London and M. Shaw, An introduction to the construction and verification of Alphard programs, *IEEE Trans. Software Engrg.* **SE-24** (1976) 253–265.